ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADUADO EN INGENIERÍA DEL SOFTWARE

Extensión de USE para soportar incertidumbre de medida en los datos primitivos OCL/UML

USE extension to support measurement uncertainty in OCL/UML primitive datatypes

Realizado por
Víctor Manuel Ortiz Guardeño

Tutorizado por
Antonio Vallecillo Moreno
Lola Burgueño Caballero

Departamento
Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA
MÁLAGA, JUNIO DE 2019

Fecha defensa:

Fdo. El/la secretario/a del Tribunal

Resumen

En la actualidad, está surgiendo la necesidad de modelar sistemas que traten con factores físicos. Con la llegada del Internet de las cosas [IoT] hay cada vez más sistemas software conectados a sensores que no dan una medida exacta. Esto puede suponer un problema al no disponer de herramientas que tengan en cuenta este posible error en la medida.

Actualmente, el modelado y diseño de un sistema se hacen principalmente con el lenguaje UML, junto con OCL para definir el conjunto de restricciones que el modelo debe cumplir. Estos lenguajes tienen tipos de datos que no tienen en cuenta esta incertidumbre de la medida. Pero también sería conveniente que, al realizar operaciones con los tipos de datos, se propague esta incertidumbre, permitiendo dar resultados a las preguntas del diseñador sobre el sistema. Además, esta propagación debería realizarse de manera transparente para el diseñador; simplemente todas las operaciones se realizarían mientras se modifica el modelo.

Para ello, se ha desarrollado una extensión de la herramienta de diseño USE. Con ella, se pueden definir los modelos UML y especificar las restricciones del modelo en el lenguaje OCL, permitiendo también la simulación de instancias del modelo teniendo en cuenta la incertidumbre de la medida, de modo que el diseñador podría ver cómo se comporta el sistema incluso antes de su fabricación.

Keywords: USE, UML, Medida, Incertidumbre, OCL

Abstract

Nowadays, the need to design systems that interact with physical devices is evident. For example, with the arrival of the Internet of things many software systems are connected with sensors. The problem is that sensors do not provide exact measures; they are normally subject to uncertainty. This can be a problem if the designer does not have tools capable of handling the possible measurement errors.

Current models and designs of software systems are developed using the UML and OCL notations. In particular, OCL is a formal language used to specify the integrity constraints of the system, as well as the rules it should comply with. These languages provide primitive data types, but unfortunately they cannot handle the measurement uncertainty associated to the values of attributes that represent physical quantities. They cannot propagate this uncertainty through the datatypes operations, either, something that needs to be done manually by the system modeller.

To tackle this problem, this project presents an extension of USE tool that we have developed for defining models in UML and OCL that support the specification and simulation of models instances with uncertainty measurement. In this way, the designer can understand how the system works with this kind of uncertainty before it is implemented.

Keywords: USE, UML, Measurement, Uncertainty, OCL

Índice

1

Introducción

Este capítulo servirá para contextualizar el entorno en el que se ha extendido la herramienta de USE. Para ello, se mostrarán los apartados motivación, objetivos del proyecto, tareas a desarrollar, metodología, fases de trabajo y por último veremos de forma general la estructura de esta memoria.

1.1 Motivación

Por parte del autor, que disfruta diseñando modelos de sistemas, encontraba este proyecto muy interesante. El hecho de poder modelar sistemas que estén más cercano a una situación real. En la que los sensores no marcan un valor exacto, sino más bien una aproximación al mismo. A parte del interés del autor por este proyecto, hay muchas partes donde este trabajo se podría aplicar.

Por ejemplo, con el crecimiento del internet de las cosas [IoT] o sistemas físicos [CPS] son ejemplos de casos en la vida real que tienen que lidiar con el problema de tratar con un sistema físico. Por ejemplo, podríamos querer modelar un sistema en el cual un robot tiene que llegar del punto A al punto B. El robot tendrá varios sensores que utiliza para ser consciente de su entorno, por supuesto, estos tendrán un margen de error. Con trabajos como este, podríamos ser capaces de predecir dónde pudiera acabar este robot desde una fase de análisis. Resulta atractivo predecir el comportamiento del mismo desde una fase tan temprana.

1.2 Objetivos

El objetivo principal de este proyecto es el de extender la herramienta USE para incorporar soporte en los datos primitivos para la incertidumbre de medida. Tanto por la interacción del usuario con la interfaz del programa, como desde los ficheros que éste consume para la definición de los modelos. Para ello, se tendrá que extender la gramática OCL para incorporar los tipos de datos *UReal*, *UInteger*, *UString* y *UBoolean* que corresponden a supertipos de los tipos primitivos *Real*, *Integer*, *String* y *Boolean* respectivamente, ya existentes dentro de la gramática.

Para extender estos tipos de datos, siempre se tendrá que permitir la interoperabilidad entre ellos. Es decir, se deberá permitir que todas las operaciones se puedan realizar entre los tipos de datos *UReal*, *Integer*, *Real*, *UInteger*. Si se hace una suma de dos tipos de estos, el resultado será del supertipo común. Por ejemplo : "1 + UInteger(2, 3) - UReal(2, 3)" tendrá como resultado un valor de tipo *UReal*. Lo mismo ocurrirá con *String* y *UString*, y para *Boolean* y *UBoolean*. Se deberá de implementar todas las operaciones recogidas como requisito en el anexo.

También se deberá dar soporte para operar con estos tipos de datos a las colecciones de OCL. Para ello, se tendrá extender la funcionalidad de los cuatro tipos de colecciones que hay, *Sequence, Bag, OrderSet* y *Set*.

Además de extender la herramienta, se tiene que llevar un proceso de pruebas y corregir, en la medida de lo posible, una librería ofrecida por los tutores para el manejo de la incertidumbre. Para ello, se realizarán pruebas unitarias, de sistema y de regresión.

A parte de los objetivos del producto resultante de este proyecto fin de carrera, se tienen como objetivo que el autor profundice en su conocimiento sobre pruebas, aprenda los conceptos sobre compiladores e intérpretes. Ya que en ingeniería del software no está esta asignatura. Como objetivo más ambicioso y útil podría ser el de modificar un software ya existente para adaptación a una situación distinta de cuando se creó. Este, es un objetivo muy útil, porque se piensa que un programador estará más tiempo manteniendo software de otras personas, que creando nuevo en su vida profesional. Por lo que se adquirirán cualidades que serán de mucha utilidad fuera de la universidad.

1.3 Tareas a desarrollar

A continuación, se desglosarán las tareas que se plantearon para hacer para conseguir los objetivos de este proyecto:

Tarea 1 Modificar la gramática de USE para permitir los nuevos tipos de datos y sus operaciones. En esta tarea se incluye el estudio del *framework* ANTLR para la construcción de lenguajes y el estudio de las gramáticas de USE.

Tarea 2 Implementar los nuevos tipos en USE. Para ello se requerirá de un estudio de cómo están implementados dentro de USE y su implementación posterior.

Tarea 3 Implementar las operaciones para los tipos con incertidumbre. Como anteriormente, se requerirá de un estudio y su implementación posterior.

Tarea 4 Diseño, especificación y ejecución de pruebas para los tipos de datos y operaciones.

1.4 Metodología

Para el desarrollo del proyecto se ha llevado una metodología iterativa incremental, trabajando sobre un subconjunto de tipos de datos y sus operaciones. Debido a la complejidad de modificar un software no propio, se escogerán los tipos de datos más sencillos para las fases más tempranas. Con la idea de ir incrementando la complejidad de los datos conforme se valla adquiriendo más conocimiento de cómo está hecha la herramienta.

Como el proyecto estaba muy centrado en pruebas, se impuesto una metodología orientada a las pruebas [TDD]. Puesto que es un aspecto muy importante a tener en cuenta para este proyecto y encaja a la perfección con la metodología iterativa incremental. En el desarrollo dirigido por pruebas (*Test Driven Development*, TDD de aquí en adelante) se escriben los casos de prueba antes de su implementación. De modo que se adquiere un conocimiento mejor del problema y como resultado, genera un código de mayor calidad.

El ciclo de trabajo de la metodología TDD implica el escribir los casos de prueba, implementarlos, implementar la funcionalidad, ejecutar todos los casos de prueba y

refactorizar. Si al ejecutar los casos de prueba se encontrara algún error, se corregiría y se ejecutaría de nuevo todas las pruebas.

La metodología iterativa incremental que se ha escogido ha sido SCRUM. Como en el desarrollo del proyecto sólo ha habido una persona, no se han usado características de SCRUM como la asignación de roles. Pero el cómo esta metodología organiza el trabajo ha sido de gran utilidad, tanto para saber qué se tiene que hacer para la siguiente versión, o para tener una visión global de en qué estado se encuentra este.

Durante el desarrollo del proyecto se ha gestionado las versiones mediante Git, y se ha alojado el proyecto en la plataforma GitHub. Las versiones del proyecto han seguido la notación *x.y.z*. Donde *x* representa la versión más alta compatible, *y* la versión más baja compatible y *z* el parche de la versión. Se escogió esta notación porque está muy estandarizada. Aunque en este proyecto *x* siempre vale 0 ya al ser software libre, el software está sujeto a cambios. La variable *y* ha sido incrementada cada vez que se realizaba un requisito, quedando como versión final del proyecto 0.120.0.

La gestión de incidencias, preguntas o mejoras también ha sido gestionada por la plataforma Github. Cada vez que se encontraba una situación extraña, se tenía alguna duda o algún aspecto se podría mejorar de alguna forma, fue reportado como *issue* en la página del proyecto. Esta forma de llevar las incidencias fue de gran utilidad tanto en la comunicación entre el autor y los tutores del proyecto, como para llevar un histórico de qué problemas nos hemos encontrado a lo largo del desarrollo del proyecto.

1.5 Fases de trabajo

Las fases de trabajo han sido organizadas por orden de dependencias. Estas son :

Fase 1

Investigación del ámbito del problema. En esta fase se estudió el código de USE, de hecho, se ha dedicado un apartado en esta memoria para esta fase. Además, se investigó cómo funciona el compilador ANTLR, puesto que era un conocimiento nuevo para el autor.

Fase 2

Desarrollo de la extensión. En realidad, esta fase se desglosa en muchas otras. Debido a que aquí se incluye el ciclo de software de análisis, implementación y pruebas de cada uno de los requisitos del proyecto.

Fase 3

Informe final y conclusiones. En la fase final, se desarrollará la memoria y los documentos necesarios para su defensa. Así como una opinión de autor sobre el proyecto, una vez haya finalizado.

1.6 Estructura de la memoria

Esta memoria ha sido escrita en el orden en el que se construyó el software. Por lo tanto, se podrá observar que el orden de los capítulos es muy similar a las tareas descritas anteriormente. A continuación, se mostrará un breve resumen de qué nos deparará cada capítulo de esta memoria.

Capítulo 2

Tecnologías utilizadas. En este capítulo introduciremos qué tecnologías se han usado en la extensión de la herramienta. Como veremos, muchas estaban ya impuestas en la construcción de esta.

Capítulo 3

Estudio de USE. Antes que nada, se tuvo que hacer un estudio de cómo funcionaba USE internamente con el objetivo de extenderlo. En este capítulo, mostraré los conocimientos que he adquirido de la herramienta y se expondrán las pautas necesarias para entender el siguiente capítulo.

Capítulo 4

Diseño de la solución. Con el conocimiento que se adquirió en el capítulo anterior, se muestra el procedimiento que se ha seguido y los problemas que se han encontrado para extender la solución.

Capítulo 5

Validación y Pruebas. Aquí hablamos de la organización que se ha seguido por las pruebas, que alternativas había y qué estrategias se han seguido.

Capítulo 6

Conclusiones. Como aporte final, explicaré mi opinión de qué he aprendido con este proyecto y cómo se podría extender.

Anexos

Como anexos se incluirá una guía de instalación, la lista de requisitos inicial y la documentación de los nuevos tipos y sus operaciones.

2

Tecnologías utilizadas

En este capítulo hablaremos de las tecnologías que se ha usado a lo largo del proyecto. Especificaremos cuáles son y para qué sirven.

2.1 Java

Es un lenguaje de programación orientado a objetos, que tuvo como objetivo ser multiplataforma. Para ello, los programas escritos en java no se compilan directamente a código máquina. Si no que más bien, compilan a un fichero intermedio, cuyas líneas de código se les llama *bytecode* y se ejecutan en una máquina virtual. Para cada plataforma en el que se quiera ejecutar un programa Java, tendrá su propia máquina virtual específica para ello, y al estar el código compilado en el mismo lenguaje (*bytecode*), se puede compartir el código entre plataformas.

Otra de las ventajas que tuvo Java con respecto a otros lenguajes de programación de su época fue las facilidades que ofrecía a los programadores. Por ejemplo, la máquina virtual de Java dispone de un recolector de basura. Por lo que los programadores ya no tenían por qué preocuparse de librear recursos que adquirían.

En este proyecto se utiliza la tecnología Java de forma inherente. USE está desarrollado en Java, por lo que se tenía que desarrollar en este lenguaje.

2.2 ANTLR

Es una herramienta de reconocimiento de lenguajes que permite construir analizadores (*parser*s), intérpretes y compiladores, a partir de unas descripciones gramaticales que se le ofrece a la herramienta. En estas se define la semántica del texto a leer.

Esta herramienta fue creada principalmente por Terence Parr, al ser estudiante de informática y necesitar una herramienta para construir compiladores que encajara a sus necesidades. A partir de esta herramienta, fue haciéndole modificaciones hasta darle un uso más general y convertirse en lo que es hoy en día.

El funcionamiento de esta herramienta es generar ficheros en un lenguaje de programación a partir de las descripciones textuales de la gramática. En estas descripciones también se puede incluir código del lenguaje objetivo. De este modo, la herramienta también incluirá este código en los ficheros generados. Normalmente esta especificación corresponde a la construcción del árbol AST, que corresponde la representación abstracta de la gramática. Aunque debido a que esta práctica tiende a no variar de un proyecto a otro, en las últimas versiones se ha implementado que la herramienta genere clases con la estructura del patrón de diseño *Visitor* o *Listener*. La elección de cual escoger es del programador, incluso de no escoger ninguno y generar por sí mismo el árbol AST.

En el caso de USE, utiliza ANTLR para leer todas las gramáticas que este contiene. La versión que utiliza es anterior a la actual, por lo que no estaba implementada la funcionalidad de escoger el patrón de diseño. Por lo tanto, genera el árbol AST por medio de código Java embebido en las descripciones gramaticales.

2.3 Intellij IDEA

Corresponde a un IDE (Entorno Integrado de Desarrollo) que se ha vuelto muy popular en estos últimos años. Esta herramienta fue creada por la empresa Jet Brains. Esta es una de las múltiples herramientas que esta empresa tiene para los desarrolladores de código. En concreto, Intellij IDEA se centra en los proyectos Java. Permitiendo un entorno amigable y atractivo para todas las fases del Software. Por ejemplo, contiene una buena interacción para realizar las pruebas unitarias, o una forma sencilla de llevar las versiones de código.

La elección de utilizar esta herramienta para el desarrollo del proyecto fue del autor de este trabajo de fin de curso. Inicialmente se planteó empezar con eclipse, pero debido a las buenas críticas que se encontró de esta herramienta, y que al ser estudiante la teníamos gratuita, me llevó a utilizarla.

Como experiencia, puedo decir que las funcionalidades que ofrece me ayudaron mucho para el estudio del código de USE. Puesto que permite extraer los diagramas de clases de los paquetes, hecho que fue muy útil para analizar la estructura y las dependencias de las clases. Muchos de las figuras que se incluyen en este proyecto están generadas con esta herramienta.

2.4 JUnit

En USE se realizó las pruebas con este *framework*. Es uno de los más ampliamente utilizados por los desarrolladores, porque casi todos los entornos de desarrollo incluyen *plug-in* para ejecutar las pruebas.

Este *framework* es muy útil para realizar las pruebas unitarias de las aplicaciones de Java. Aunque también permite realizar pruebas de regresión sobre los casos de prueba. Hecho que es muy útil en un desarrollo en el que se involucra a las pruebas de manera iterativa.

2.5 Git

Git es una herramienta para el control de versiones. Fue desarrollada por Linus Torvalds para dar solución a la manera de trabajar con proyectos de software libre. Donde cada programador aporta lo que puede y cuando puede. La herramienta permite compartir el código entre varios programadores y es capaz de gestionar las modificaciones que se hacen sobre el código fuente.

Para llevar un control de versiones de este proyecto se optó por esta tecnología. Ya que es ampliamente usada y nos era bastante familiar.

2.6 Apache Ant

Apache Ant es una herramienta de línea de comandos que nos permite configurar cómo se construye un proyecto. Dentro de las funcionalidades que podemos utilizar de esta herramienta son; compilar el proyecto, empaquetarlo, ejecutarlo o pasar las pruebas, entre otras.

La configuración de los proyectos se realiza en un fichero *build.xml* y en el podemos establecer también objetivos personalizados. Por ejemplo, como veremos más adelante, la construcción de las gramáticas está hecha en tiempo de compilación mediante Ant.

En este proyecto hemos utilizado Apache Ant de forma inherente. El proyecto antes de pasar por nuestras manos, ya se compilaba con esta tecnología.

3

Estudio de USE

En este capítulo pondremos en contexto cómo está hecho USE y qué estructura interna dispone esta herramienta. Explicar todo USE no es el ámbito de este trabajo, puesto que es una herramienta bastante extensa. De modo, que nos centraremos en lo necesario para desarrollar los objetivos.

Los objetivos principales que abordaremos en este capítulo serán averiguar dónde se puede cambiar las gramáticas de USE; dónde se puede definir nuevos tipos, su valor y sus expresiones; y, por último, dónde están situados las pruebas unitarias y de sistema dentro de USE.

Sin embargo, será necesario un apartado para hacer un recorrido rápido por las carpetas de USE, presentando dónde están todos los paquetes y para qué sirven.

3.1 Vista general

Al abrir el directorio de USE encontraremos los ficheros listados en la Figura 1. Al ser un software de código libre podemos ver que el código fuente (contenido en el directorio *src*) se distribuye junto a su ejecutable (contenido en *bin*), y su licencia en el fichero COPYING. Los creadores dispusieron de varios ficheros para dar información sobre la herramienta y ejemplos de uso. Los ficheros o directorios que contienen esta información son ChangeLog, NEWS, README, README.OCL, INSTALL y examples.

En la carpeta del código fuente se encuentran los 4 paquetes raíces de USE. Los paquetes son *gui, runtime, test* y *main*. Cada paquete corresponde a encapsular la implementación de la interfaz gráfica, el funcionamiento de la consola de comandos y los

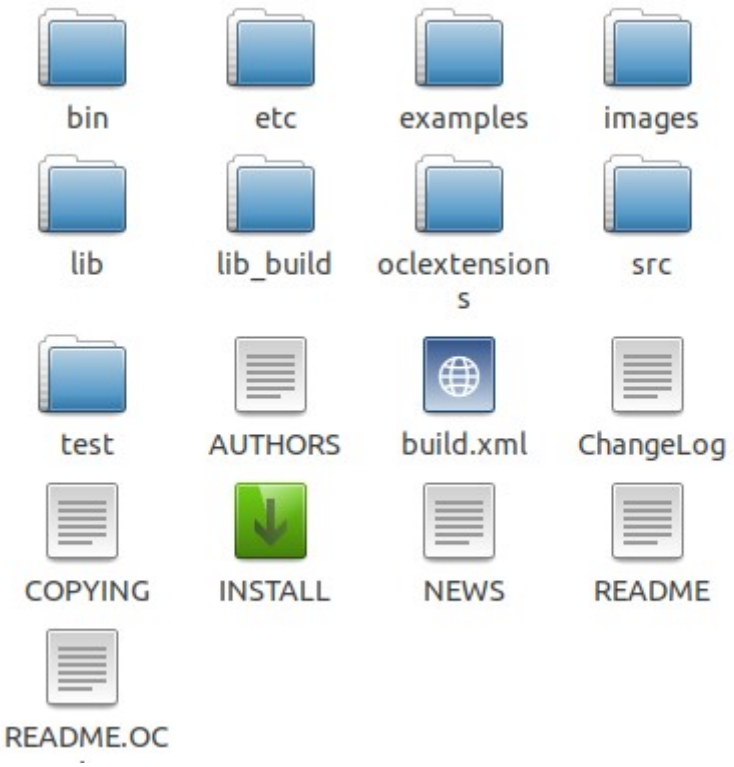

Figura 1: Ficheros contenido en la raíz de USE

plugin, agrupar las pruebas de sistema; y contener toda la lógica de USE, respectivamente.

En este proyecto se ha modificado el paquete *main* que es el que contiene la definición de OCL y el paquete *test*. No profundizaremos en los paquetes *gui y runtime*.

Paquete main

Lo más importante de este paquete para este proyecto es que en él se encuentran las gramáticas y la definición interna de los tipos, valores y expresiones. En este paquete se encuentran una variedad de paquetes como se observa en la Figura 2. A pesar de ello, los paquetes importantes son solo dos, *uml* y *ocl.*

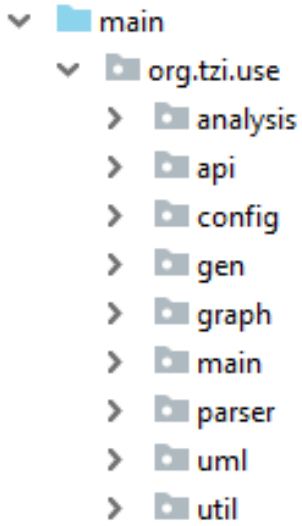

Figura 2: Listado de paquetes debajo de main

Dentro del paquete *uml* tenemos el código que da estructura el modelo, al sistema que representa el modelo y a las expresiones OCL. Aquí podemos encontrar el código dónde se definen los tipos, valores y expresiones (Figura 3). Entraremos en profundidad en cómo está hecho en este mismo capítulo.

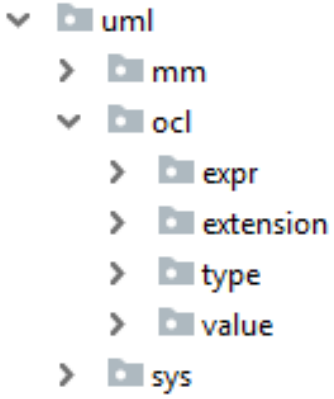

Figura 3: Paquetes dentro de org.tzi.use.uml

Paquete test

El paquete *test* contiene todas las pruebas unitarias de todo USE. Están organizadas por paquetes con el nombre correspondiente a la clase que prueban. Como la herramienta ya tiene un tiempo y JUnit ha avanzado mucho, el modo de hacer las pruebas según la versión 3 de JUnit. De modo, que en cada paquete se encuentra una clase llamada *AllTest* que crea una *Suite* y las ejecuta. En el paquete raíz se encuentra la clase que va llamando a todos los grupos de prueba.

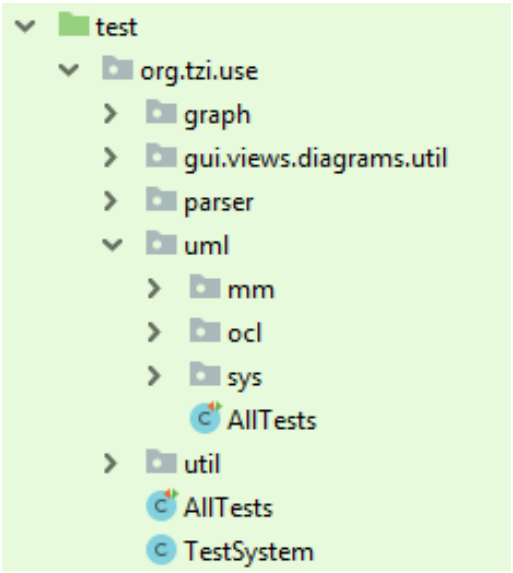

Figura 4: Listado de paquetes y clases del paquete raíz test

Parte de las pruebas de sistema también se encuentran en este paquete. Para ello, los creadores de USE hicieron pruebas unitarias del compilador de USE mediante ficheros. Estos ficheros están organizados por función abstracta sobre la gramática que se quiere probar. Por ejemplo, "creación de tipos básicos", "operaciones con tipos básicos", etc. Las pruebas de sistema se encuentran en el paquete *org.tzi.use.parser*.

3.2 Gramáticas

Las gramáticas están definidas con la tecnología ANTLR en el paquete *org.tzi.use.parser*. En él se encuentran las gramáticas siguientes:

1. OCL (*Object Constraint Lenguage)* es el lenguaje formal, usado para describir expresiones dentro de modelos UML. Estas expresiones normalmente son invariantes que deberá de cumplir el modelo, o consultas de este. También es posible usar OCL para definir acciones sobre el modelo [OCL].

Una de las características importantes de OCL es que la ejecución de expresiones que definan un método no puede tener efectos laterales. Esto es interesante para este proyecto porque se han extendido los tipos de OCL.

2. SOIL (*Simple OCL-like Imperative Language)* es un lenguaje propuesto por los creadores de USE. Esta gramática depende de OCL, puesto que su finalidad es la de no alterar su esencia. SOIL define una gramática imperativa, que junto a OCL permiten definir prácticamente cualquier método del sistema, al nivel de modelo. Permitiendo simularlo y ver cómo se comporta [SOIL].

3. USE (*UML based Specification Enviroment*) define una gramática para especificar sistema basado en UML. Permite definir la estructura del sistema, sus invariantes, precondiciones, etc.
   Esta gramática depende de OCL y SOIL debido a que todas las condiciones que se pueden definir en el modelo se hacen en OCL, y si se quiere establecer el comportamiento de algún método se deberá usar SOIL.

4. SHELL. USE define una gramática para interacción con el usuario por consola de comandos. Por tanto, aquí se define las posibles operaciones desde la consola de comandos. Esta gramática depende de OCL y SOIL, puesto que permite que el usuario haga consultas OCL del estado del sistema, o ejecute algunas operaciones SOIL.

5. TESTSUITE. Existe una gramática dentro de USE que define el lenguaje en él se escriben las pruebas de sistema. En esta gramática se define como la persona que desarrolla las pruebas debería de indicar el caso de prueba, y qué debería de dar USE como salida por consola.

La gramática de TESTSUITE es muy interesante para este TFG puesto que las pruebas son una parte muy importante del mismo. El hecho de ver que en un proyecto se definan de esta forma las pruebas, ha servido para darme cuenta de que las gramáticas se pueden llevar al ámbito de las pruebas para hacerlas más fáciles.

Estoy a favor la práctica de llevar las pruebas a la gramática, porque está relacionado con la razón por la que el creador de ANTLR creó su herramienta: para hacer más fácil la

interacción entre el humano y la máquina. Dicho autor pensaba que las gramáticas ayudan mucho a este fin, y que todo el mundo debería usarlas. [ANTLR]

Sin embargo, hacer las pruebas de esta forma tiene ventajas y desventajas. La principal ventaja es que cualquiera puede ver el fichero de pruebas y es menos complejo que ver un código en Java o en cualquier otro lenguaje. La visión que se tendrá de este fichero es clara y se podrá extender las pruebas de una forma más fácil y con muchas menos líneas de código.

Por otro lado, al ejecutar estas pruebas se tendrá menos *feedback* que las hechas con la tecnología Junit. Esto es debido a que, si se intentan ejecutar 200 pruebas por ambos métodos, con la gramática se obtendrán menos pruebas inválidas que con Junit. La razón es simple: el primer método (usando una gramática) lee un fichero de pruebas y las va ejecutando secuencialmente, de modo que cuando encuentra una que falla, aborta la ejecución y deja las demás sin ejecutarse. De este modo el programador tiene que corregir el error y volver a ejecutar las pruebas. Probablemente, fallará otra prueba y el programador estará en este ciclo hasta que todo funcione correctamente.

Por el contrario, si con Junit se agrupan bien las pruebas, pueden fallar todas a la vez, y de esta forma el programador podrá corregir todos los errores y volver a ejecutar las pruebas.

Todo esto depende de la forma de agrupar las pruebas: si cada caso de prueba se encapsula dentro de un método de Junit, se tendrá un buen *feedback*. Lo equivalente con la gramática sería tener cada caso en un fichero, lo que puede ser tedioso y resta cierto interés al caso de la gramática según mi punto de vista.

Concluyendo, ¿qué hemos hecho en la extensión de USE? Nuestra solución ha consistido en utilizar ambas técnicas. Este tema será tratado en el capítulo de validación y pruebas en la sección de estrategias.

Todas las gramáticas son dependientes en el orden en el que está descritas. Por ejemplo: SOIL depende de OCL, USE depende de OCL y SOIL, etc. Parece interesante la construcción de estas dependencias en tiempo de compilación. La figura 4.1 muestra cómo se construyen las gramáticas. Esto se realiza en tiempo de compilación mediante Apache Ant definida en el fichero *build.xml*, en la regla *create-parser*.

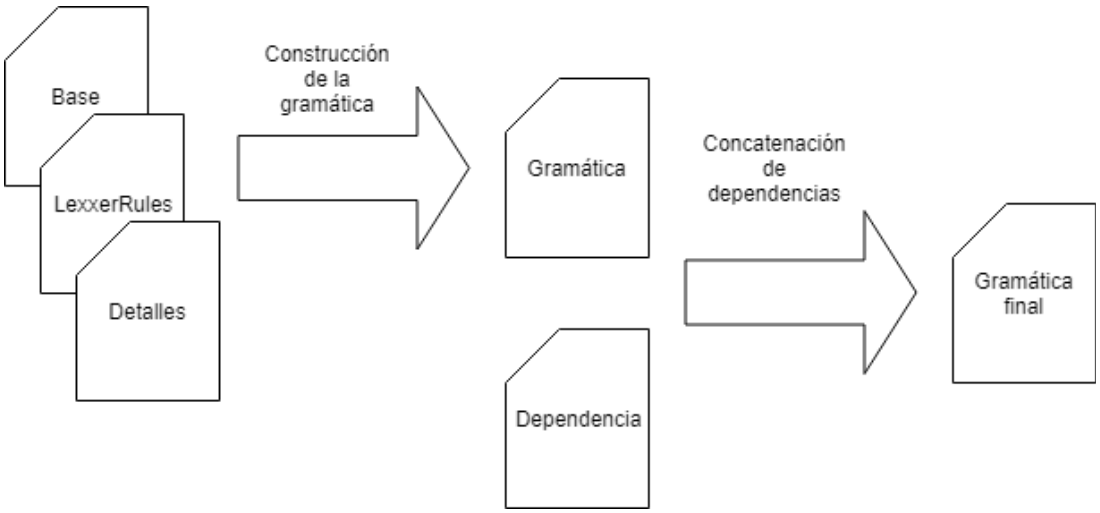

Figura 4.1 Abstracción del flujo de la construcción de gramáticas en USE.

Una gramática se construye a partir de un fichero base, literales de la gramática (LexxerRules en la figura) y unos detalles. En el fichero base se definen todas las reglas de la gramática. En el segundo están todos los literales, tales como números enteros, identificadores, etc.; estos son compartidos por todas las gramáticas en USE. Por último, un fichero al que hemos decidido llamarle "detalles" porque contiene el código Java para enlazar los objetos en los que se referencia la gramática, debido a que estos ficheros están en varios paquetes.

Una vez que se construye una gramática en el paso anterior, se le añaden las gramáticas de dependencia, se concatenan todas y se invoca al generador de gramática de ANTL para construir todas las clases Java de compilación.

Este proceso es útil porque si se quiere cambiar una gramática, solamente habría que tocar el fichero base y cambiar las reglas. En tiempo de compilación se construirán las gramáticas de nuevo, aplicando los cambios hechos en todo USE.

Implementación y funcionamiento

Actualmente ANTLR puede generar una estructura de clases que se basa en el patrón de diseño *Listener* o *Visitor* (dependiendo de cómo prefiera definirlo el programador), con el fin de ahorrar tiempo al programador y que sólo lo dedique a rellenar el comportamiento de la gramática.

USE crea una representación intermedia de la gramática. Esta representación se utiliza comúnmente al enfrentarse a gramáticas que definen un lenguaje de programación, ya que estás gramáticas suelen ser más complejas que otras y requieren de un tratamiento especial. Esta representación intermedia se llama AST (*Abstract Syntax Tree)* y se utiliza mucho para crear compiladores.

En la implementación de USE todos los nodos de la representación de la gramática heredan de la clase AST, y con esto se crean las dependencias que vemos en la Figura 5. Dentro de cada sub-paquete, que corresponde a una gramática, se encuentra la especificación del comportamiento particular de cada gramática. Por ejemplo, en OCL la clase base de los nodos se llama *ASTExpression* y todos los nodos de OCL heredan de ella.

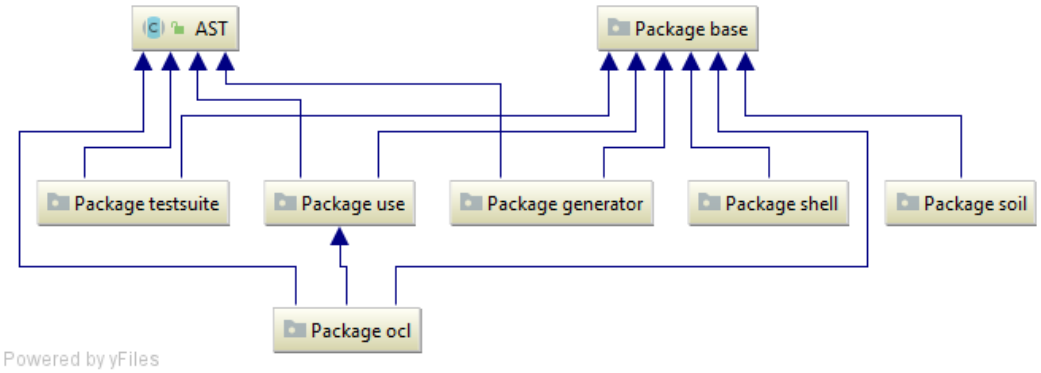

Figura 5: Dependencias de paquete parser y sus gramáticas

3.3 OCL

En el este apartado se refleja las preguntas e inquietudes que tuvimos al empezar con el proyecto, a cerca de la extensión de tipos, valores y expresiones de OCL en USE.

Al principio, nos preguntábamos dónde estaba esta parte del código en USE. Este código se encontraba en el paquete *org.tzi.use.uml.ocl*, y este contenía cuatro paquetes más; *type, value, expr* y *extensión*. Dónde se encontraba todo el código que estábamos buscando. A excepto del paquete *extensión* que no se ha usado en todo el proyecto, por ser la medida que tiene use de extender los tipos de manera externa.

Estos tres paquetes restantes son los que nos interesaban en nuestro estudio. Nos interesaba saber cómo se creaban tipos, valores o expresiones, de modo que analizamos las dependencias entre ellos. La Figura 6 muestra las dependencias y se puede observar que tanto valores como expresiones dependen de tipos y las expresiones dependen de los tipos y de los valores. De modo que, para construir un tipo nuevo con todas sus operaciones, como puede ser UReal, es necesario seguir este orden de dependencias. Primero se crearía el tipo, luego el valor y por último todas las expresiones.

Para cada una de las partes habladas anteriormente, se ha dedicado un subapartado para hablar más detenidamente de cómo está hecho.

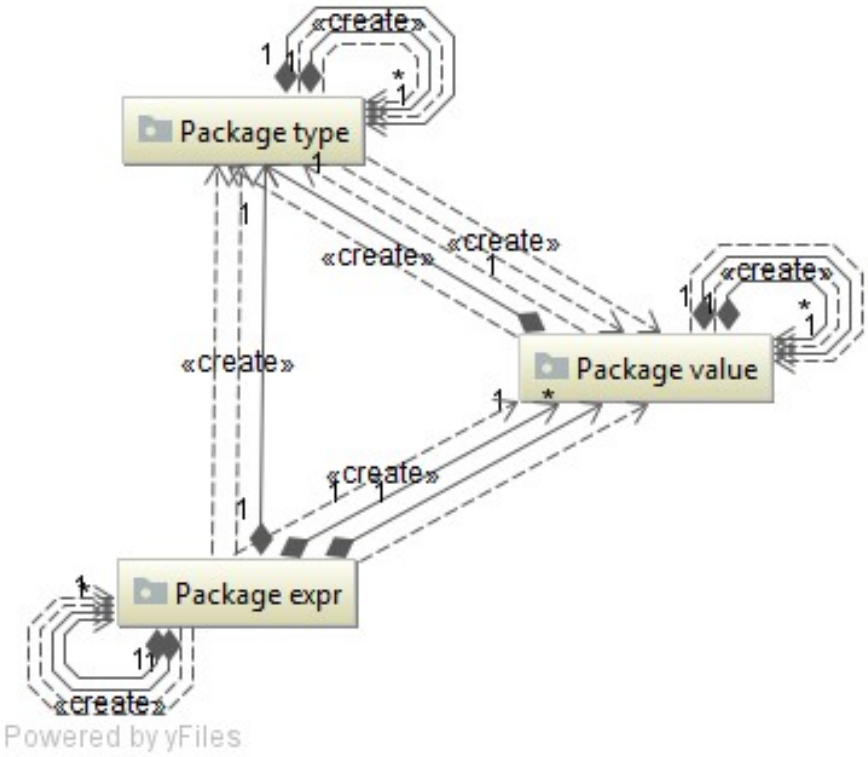

Figura 6: dependencias entre tipos, valor y expresiones OCL

Tipos

Todos los tipos tienen como super tipo a la interfaz Type. Desde esta interfaz se hereda toda la jerarquía de tipos de OCL en USE, como se puede ver en la Figura 7. No solo los tipos de OCL dependen de la interfaz *Type*, por ejemplo, las clases definidas también son tipos. Por lo cual, de este paquete depende muchos otros dentro de la herramienta y define las bases de OCL.

Para crear un nuevo tipo sólo hay que crear una clase y que derive, directa o indirectamente, de la clase *Type*. Por ejemplo, en la figura anterior vemos que el tipo Booleano deriva de *BasicType*, este de *TypeImp* y este último de *Type*.

Es interesante el por qué está el *TypeImpl,* ya que la interfaz *Type* define muchos métodos, tediosos para el programador que quiera extender la funcionalidad, por lo que los creadores de USE pusieron esta clase intermedia para simplificar las clases hijas, hecho que he encontrado útil en la extensión de la herramienta.

La interfaz *Type* define el comportamiento básico de un tipo, contiene los métodos para saber si un tipo es subtipo de otro, si es super tipo, etc. Entre las cosas que define, está el saber si una instancia es tipo de *Real*, *String* u otro. Por defecto, la clase *TypeImpl* define que

toda instancia que se derive no es de ningún tipo. Para que más adelante, el programador sobrescriba estos métodos estableciendo el comportamiento deseado. Por ejemplo, la clase *RealType* sobrescribe los métodos *isTypeOfUReal()* y *isKindOfUReal()*.

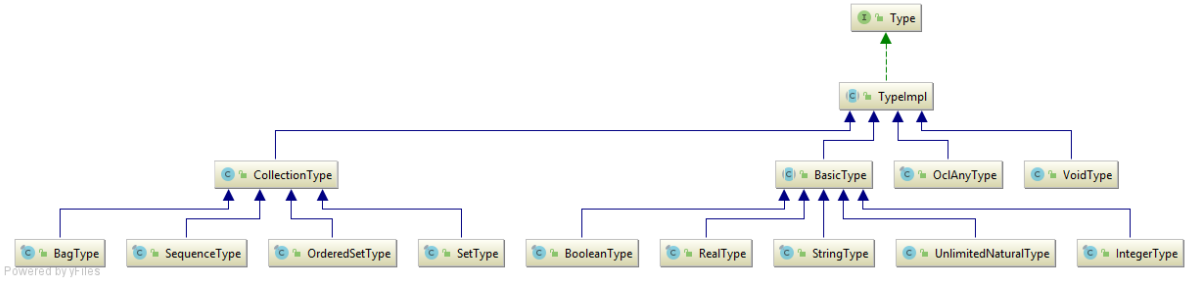

Figura 7: Jerarquía de tipos OCL

Un apunte interesante sobre la implementación de tipos es la creación de estos. No se crean dos instancias del mismo tipo. Para ello los creadores de la herramienta utilizaron el patrón de diseño *Factory*. En el cual los constructores de los tipos tienen visibilidad de paquete y la clase factoría crea las instancias. La clase factoría de tipos se llama *TypeFactory* e implementa que sólo se pueda crear una instancia. Mediante el patrón *Singleton*.

Valores

Los valores tienen una relación de composición con su tipo. No se puede crear un valor sin su tipo y no tiene sentido que exista un valor que no esté tipado. Por lo tanto, todo valor dentro de la herramienta deriva de la clase abstracta *Value*, cual tiene un constructor con un tipo como argumento.

De este modo, se puede asignar valores a varios tipos. Es decir, se puede definir un valor que fuera *Integer* y asignárselo a un tipo *Real*. La relación que existe entre el valor y el tipo es de muchos a muchos. Más adelante, la herramienta se encarga hacer verificar que estas asociaciones sean correctas. No se debería de asignar un valor de tipo *String* a un *Real.*

Para resolver este problema, se delegó esta responsabilidad en las clases del apartado anterior, los tipos. En la interfaz *Type* se definen método para conocer cuáles son los super tipo y los subtipos. Por lo tanto, si el tipo de un valor no estuviera contenido en el conjunto de los subtipos, se lanzaría una excepción y se detendría el flujo normal del programa. Como apunte, todo tipo está incluido en el conjunto de sus subtipos.

Por el hecho anterior, la clase abstracta *Value* define métodos para conocer el tipo del valor. Además de ello, implementa la interfaz de Java *Comparable* que junto a los métodos por defecto *equals* y *hashcode*, juegan un papel curioso en algunos casos. Por ejemplo, 1 debe

ser igual a 1.0, y más adelante en esta memoria : 1 = 1.0 = *UInteger*(1, 0) = *UReal*(1, 0). De este caso hablaremos en la parte del diseño de la solución.

En la Figura 8 podemos ver la jerarquía de valores de OCL, que como es de esperar se parece mucho a la jerarquía de tipos.

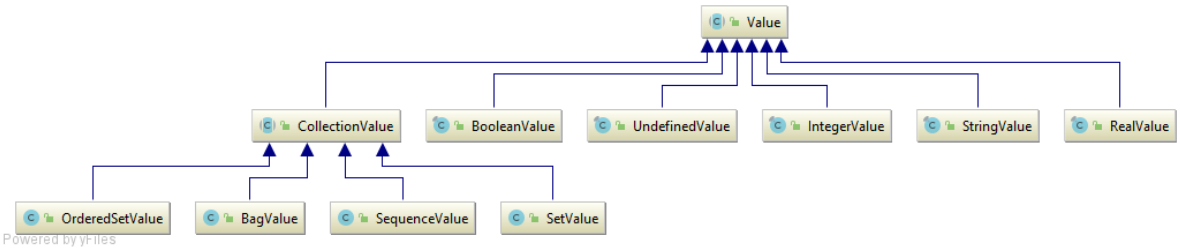

Figura 8: Jerarquía de valores en OCL

Una cosa interesante que he visto es la creación de valores. A priori, no debería de haber nada extraño en la creación de estos. Pero ¿se debería de controlar la creación para instancias de *Value* con el mismo estado? Al parecer, los diseñadores pensaron que depende. Por ejemplo, para reales no, supongo que por que la parte decimal puede variar mucho de un real a otro. Pero para enteros decidieron controlar este comportamiento, y lo hemos encontrado muy interesante. Han controlado que si se crea un entero en el rango de [0-20] se devuelve una instancia única. Suponemos que los diseñadores pensarían que estos valores son los más usados por los usuarios, ya que este comportamiento se extiende también a las multiplicidades de las asociaciones. Que tienden a ser números pequeños. Esta decisión optimiza la memoria y la carga de creación de nuevos objetos en la máquina virtual de Java.

Expresiones

Las expresiones en de USE son muy variadas, para cada nodo de la gramática correspondiente existe una expresión. Aunque en este apartado nos centraremos en las expresiones de OCL. En las cuales hay dos puntos principales en los que se focalizó el estudio de la herramienta, ¿cómo están hechos los literales de la gramática?, ¿cómo están hechas las operaciones estándar?

En la primera pregunta nos referimos a literales a toda constante de un tipo de datos. Por ejemplo, la expresión entera constante “1”, que tendría como valor 1. En la segunda pregunta se quiere conocer dónde están las operaciones estándar de los tipos de datos, como la suma, multiplicación, etc.

Antes de abordar estas preguntas, es conveniente dar una visión general del paquete en el cual se encuentran las expresiones de OCL (*org.tzi.use.uml.ocl.expr*). En él se encuentran muchas clases, y prácticamente todas heredan de la clase abstracta *Expression*. Con un punto

de vista abstracto, de una expresión se quiere conocer su tipo, su posición en la lectura de esta (por motivos de depuración e informe de errores) y la lógica de evaluación de esta. Siguiendo con el ejemplo del párrafo anterior, la clase responsable de evaluar una constante entera es *ExpConstInteger* que deriva de *Expression*.

En el caso de las expresiones, no hay una jerarquía como en los tipos o valores. En las que se agrupan según ámbito. Más bien, tienen una jerarquía "plana", en el sentido que no hay más de dos niveles en el árbol de herencia, teniéndose como cumbre a la clase *Expression*.

Con la explicación anterior, queda prácticamente resuelta la primera pregunta que nos hicimos al principio; ¿cómo están creadas las constantes? Simplemente, tiene que existir su tipo y un valor que lo represente. Con estos dos ingredientes, añadimos el tercero que sería una clase que represente la expresión constante, pero ¿eso es todo? Bueno, hasta ahora hemos estado hablando de las tipos, valores y expresiones como si estuvieran dentro de una burbuja y nada interaccionara con ellos. Así que vamos a salir de ella un poco para responder una pregunta. ¿Quién instancia estas expresiones?

Como explicamos en el capítulo de las gramáticas - implementación y funcionamiento, existe un árbol AST con los nodos de la gramática que es construido por el compilador de ANTLR. El árbol AST genera las expresiones que estamos estudiando en este capítulo. Por lo tanto, también existe un nodo en el árbol AST que representa a una constante y que, a su vez, genera dicha expresión. Estas son evaluadas en una fase posterior por un evaluador.

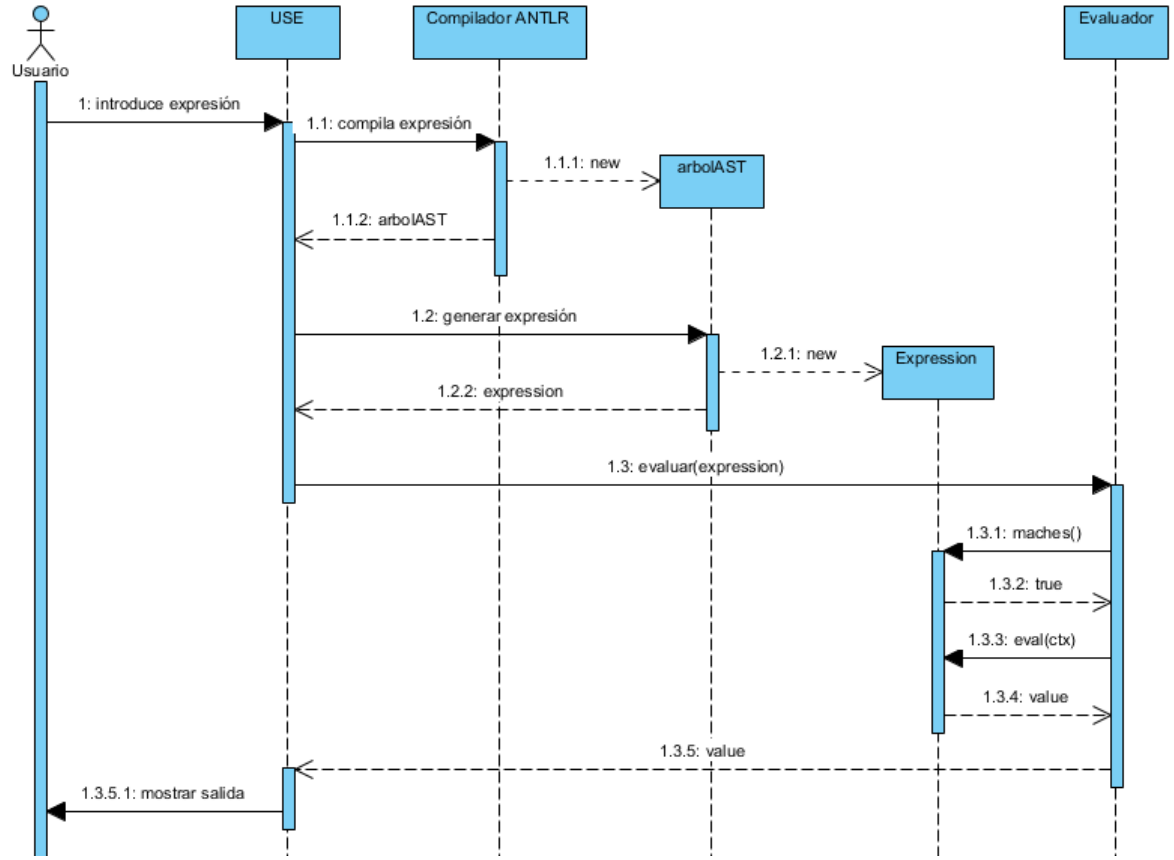

Figura 9: Diagrama de secuencia resumido de la resolución de las expresiones OCL

El segundo punto de este apartado sería cómo están hechas las operaciones estándar. Supongo que esto debió ser un buen ejercicio mental para los diseñadores en el día de su planteamiento. Porque la solución que le dieron al problema nos pareció muy interesante. Estaba el problema de la adición de nuevas operaciones en el futuro, como es uno de los objetivos de este proyecto; y la agrupación de éstas según su tipo de dato. Pues bien, empecemos a investigar cómo lo hicieron.

Para solventar el primer problema usaron el patrón *Factory*, pero esta vez distinta a las anteriores veces que hemos hablado de esta solución. Las expresiones tienen un símbolo o nombre de función. Por lo que éstas se crean con este símbolo. Por ejemplo, si queremos crear la suma, tendríamos que llamar a la clase fábrica con la cadena "+" y los dos argumentos.

Pero ¿cómo sabe cuántos argumentos hay que pasarle a cada expresión? Bueno, esto es responsabilidad de cada expresión. Cada expresión estándar hereda de la clase *OpGeneric* que se encuentra en el paquete *org.tzi.use.uml.ocl.expr.operations*. En ella se encuentran los métodos *maches()* y *eval()*. El primero comprueba el número de parámetros de la expresión y si estos son correctos. Es decir, la expresión correspondiente a la suma entera debe disponer de dos parámetros y ambos ser de tipo entero. El segundo método resuelve la operación, este recibe como parámetro el contexto del programa. En el cual se almacena información como el

número de variables que hay, al igual que sus valores. Con esta información, el programador puede resolver la operación y devolver un resultado.

Ambos métodos de los que hemos hablado anteriormente se llaman en el orden descrito. Por lo tanto, en el método *eval* el programador no tiene por qué hacer comprobaciones de tipos, porque ya se ha realizado antes.

A parte de lo anterior, esta clase la clase abstracta *OpGeneric* se encarga de registrar las nuevas operaciones. Ésta contiene varios métodos estáticos para que el programador registre sus nuevas operaciones. Para ello, tienes que especificar el símbolo de la operación y qué clase se encargará de dar servicio a tal símbolo.

Por lo tanto, todas las expresiones para *Integer, Real,* String, etc. Se encuentran recogidas en este paquete por clases estáticas, que se registran en *OpGeneric*. Todas estas se anotan en un diccionario que es consultado por la clase *ExpStdOp* al encontrar un nodo en el árbol AST con una expresión unaria, binaria, etc.

Habiendo respondido a todas preguntas que nos hicimos acerca de cómo están hechas las expresiones. Nos surgieron otras muchas a cerca de las pruebas que deberíamos de hacerle después, y decidimos hacer un estudio de los posibles errores o situaciones que podrían ocurrir. Llegando a la conclusión de que hay 4 tipos de pruebas :

1. Errores de compilación.
2. Errores semánticos.
3. Errores de formación de la expresión.
4. Errores de evaluación.

Los errores de compilación son aquellos que detecta ANTLR con ayuda de la definición de la gramática. Principalmente son errores sintácticos. También se pueden detectar errores semánticos, pero esto depende de cómo esté hecha la gramática, y se dejó libertada para controlar estos errores en otra fase de la compilación.

Los errores semánticos son tratados en cada nodo del árbol AST. Se preparó una excepción llamada *SemanticException* para que fuera lanzada cada vez que el programador encontrara una situación extraña. El evaluador informará al usuario de esta situación si se encuentra la excepción. Por ejemplo, sucede si el usuario crea una variable de un tipo que no existe en la herramienta. Si por ejemplo el usuario intentará escribir una clase de use con el atributo “ testing : Inventado”.

Los errores de formación de la expresión ya han sido mencionados en este mismo capítulo. Corresponden al número de argumentos y a su tipo. Si alguno de estos es incorrecto,

se lanzará una excepción *ExpInvalidException* y se informará al usuario. Por ejemplo, estos errores suceden cuando el usuario introduce una expresión como la siguiente : “2 / ‘tres’”, según la gramática es una operación binaria, pero según la especificación de la división entera, esto no podría suceder. Porque los dos parámetros deben de ser de tipo entero.

Por último, nos queda el error de evaluación. Estos suceden cuando ya la herramienta resuelve los valores, procede a operar con ellos y estos no son correctos. Como ejemplo está el caso clásico de “1 / 0”. Esta expresión está bien formada, pero no tiene solución real. Pero esto no es detectado hasta llegar a este paso, dado que es sintáctica y semánticamente correcta, y contiene el número correcto de parámetros y los tipos correctos. Pero el resultado sería infinito. En este caso particular se lanza una excepción *ArithmeticException* y el evaluador establece el valor de la expresión a *UndefinedValue* por defecto. Este comportamiento funciona con cualquier *RuntimeException*, por lo que el programador puede definir otras para su propósito.

3.4 Testing

La última cosa que se estudió inicialmente fue cómo se le pasan las pruebas a USE. Puesto que este proyecto sigue una metodología TDD, no podía empezar a codificar sin pasar las pruebas unitarias. Por lo que, sabiendo todo lo hablando anteriormente, nos dispusimos a resolver tres preguntas. ¿Dónde están las pruebas unitarias?, ¿dónde están las pruebas de sistema? Y ¿cómo puedo ejecutar estas pruebas?

Responderemos a las dos primeras preguntas en el primer apartado y la tercera en el segundo.

Organización de las pruebas

Las pruebas unitarias están organizadas en paquetes, cuyo directorio se encuentra en */ src/test.* De esta carpeta cuelga todos los test unitarios que los creadores hicieron. La organización de las pruebas unitarias es bastante sencilla. En cada paquete, existe una clase que construye la *Suite* de pruebas que se van a ejecutar y existe una clase que contiene las pruebas por cada clase que se quiere probar. Como ejemplo, podemos observar la Figura 10.

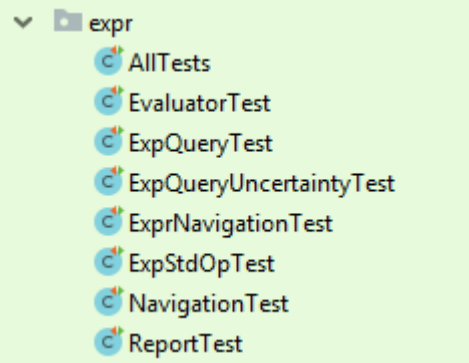

Figura 10: pruebas unitarias para las expresiones de OCL

En la figura podemos observar la clase que construye la *Suite* que es *AllTests* y las demás clases corresponden a una agrupación de pruebas para el paquete. Por ejemplo, están todas las pruebas de las operaciones estándar.

Con respecto a las pruebas de sistema, para nuestro problema, son aquellas que interactúan con el usuario directamente. Esto se puede emular probando el compilador directamente. De este modo, se tratará a la herramienta como una caja negra y sólo nos interesará el resultado. Por ejemplo, si el usuario introduce "1 + 1" lo que esperamos es "2 : Integer".

Hemos encontrado dos implementaciones de las pruebas de sistema dentro de USE. La primera mediante las pruebas unitarias y la segunda de manera externa.

Las pruebas de sistema junto a las pruebas unitarias se encuentran en el paquete del analizador. En él se encuentran también ficheros con la gramática de pruebas. La clase de pruebas lee estos ficheros y los introduce en el compilador de ANTLR, éste ejecuta las líneas del fichero, devuelve un resultado y la clase de pruebas verifica si el resultado es el esperado.

La segunda implementación está basada en la generación de ficheros mediante *scripts* escritos en PERL. Estas pruebas no se encuentran junto con las pruebas unitarias. Se encuentran en la carpeta */test* desde el directorio raíz. Las pruebas están contenidas en ficheros, y existen dos tipos: los que contienen el modelo a probar (file.use) y los que contienen la entrada (file.in). Ambos ficheros deben de llamarse del mismo modo. Cuando se inicia el script genera 3 ficheros por cada prueba, uno que contiene los comandos que se tendrán que pasar a USE (file.cmd), otro que contiene la salida esperada (file.out) y por último el resultado de la prueba (file.result). El funcionamiento del *script* es generar el file.cmd, ejecutar cada comando en la terminal de USE e ir comparando línea por línea con el fichero file.out.

Ambas implementaciones tienen sus puntos fuertes y débiles. Por ejemplo, la primera es muy fácil de depurar. Ya que al trabajar con un entorno de desarrollo integrado (IDE, de

aquí en adelante), hoy en día viene integrado con soporte de depuración por JUnit. Por otro lado, la segunda implementación es más potente por la posibilidad de cargar un modelo para las pruebas, y ejecutar comandos en ella. Pero como hemos dicho antes, un fallo es más difícil de depurar en esta implementación. Pero creo que el objetivo era hacer pruebas de forma masiva, y se ajusta muy bien a este propósito.

Ejecución

Podemos elegir qué pruebas queremos ejecutar, las unitarias, las de sistema o ambas. Para ello podemos hacerlo de varias formas:

Si disponemos abrimos el proyecto desde un IDE, automáticamente detectará las pruebas unitarias. Por lo tanto, simplemente tendremos que ejecutarlas con la opción correspondiente al IDE que se use.

Las demás opciones están escritas en el fichero *build.xml* que es ejecutado mediante ANT. Por lo tanto, escribiré los comandos que se tendría que escribir para ejecutar cada parte.

Para ejecutar las pruebas unitarias :

```
ant test-junit
```

Para ejecutar todos los test:

```
ant test
```

Estas son todas las formas de ejecutar las pruebas, sin embargo, estos se ejecutan de forma automática si se ejecuta el comando *release* de ANT.

4

Diseño de la solución

En este capítulo hablaremos de los pasos que se han seguido para cumplir los requisitos impuestos. Para ello, hablaremos de cómo se han extendido los tipos, valores y expresiones a los tipos de datos *UReal, UInteger, UBoolean* y *UString*.

Se intentará hablar en el capítulo de forma general con respecto a cada uno de los tipos que se van a introducir en USE. Excepto en algunos casos, en los que se hablará de algunas particularidades.

A lo largo del desarrollo del proyecto se han encontrado varias decisiones de diseño que han hecho impacto en la solución. Comentaremos cuáles son, qué alternativas había y por qué hemos tomado dicha solución. De hecho, el primer apartado de este capítulo está reservado para la primera decisión de diseño a la que nos enfrentamos.

4.1 Incorporación de la librería de incertidumbre.

Para el desarrollo del proyecto, los tutores facilitaron una librería con las operaciones de los tipos que se iban a implementar [ATLIB]. En ella se encuentran los tipos de datos que se tienen que introducir en USE. De modo, que el primer paso era ¿cómo introducimos estos tipos en USE?

Se nos ocurrieron dos formas de hacerlo, introduciendo directamente las clases de la librería en USE o haciendo uso del patrón de diseño *Adapter*.

La primera solución sería sencilla de implementar, simplemente copiaríamos las clases dentro del paquete de tipos OCL y se harían modificaciones necesarias para que fuera compatible con la herramienta.

La segunda solución sería crear una clase adaptadora por cada tipo nuevo de datos y dejar la librería como dependencia del proyecto. Desde nuestro punto de vista, este patrón de diseño se ajusta bien al problema. Porque lo que se quiere intentar hacer compatible la interfaz actual de la librería a la de USE. Por lo tanto, por cada clase de la librería se crearía otra dependiente de la primera.

Ambas opciones solucionan el problema, pero ¿cuál es mejor? Por un lado, la primera solución sería más rápida. Porque en los casos de uso en los que el usuario necesite operar con valores con incertidumbre se realizarán menos llamadas. Por lo tanto, menos sobrecarga en la máquina virtual.

Sin embargo, de la otra manera sería más lento, pero mucho más fácil de mantener. Porque al tener la librería de incertidumbre de manera externa y como dependencia del proyecto. Si en un futuro se quisiera modificar una operación u añadir otras, simplemente habría que sustituir el fichero de la librería por el nuevo. Con la otra opción, los futuros programadores de la herramienta habrían tenido que modificar el código de USE directamente.

De modo que tenemos en un lado de la balanza el rendimiento y al otro la mantenibilidad. Motivo de esto fue una de nuestras primeras reuniones con el proyecto en marcha. Decidimos que el rendimiento no era un objetivo importante en este proyecto y priorizamos más la mantenibilidad. De modo que entraremos en detalles de la segunda solución en el apartado de extensión de valores de este mismo capítulo.

4.2 Extensión de tipos

La creación de nuevos tipos en USE se realiza de forma muy rápida y sencilla. A diferencia de las pruebas que se le tienen que someter una vez hechos los tipos. Porque hay que estar seguro de establecer bien la jerarquía de tipos en OCL. Por ejemplo, *Integer* es subtipo de *Real* y éste de *UReal*. Este comportamiento hay que dejarlo bien definido, e implica que no solamente hay que modificar las clases que se van a introducir nuevas. Siguiendo el ejemplo anterior, para introducir el tipo *UReal* hay que modificar tanto *Integer* como *Real* para que tengan a éste como supertipo.

La estructura de clases que se han añadido para introducir todos los tipos necesarios en este proyecto están recogidos en la Figura 11: . En ella se puede observar que se ha creado una clase *UncertainType* para agrupar a estas clases con incertidumbre. Todas las clases ancestro de ésta son clases de USE.

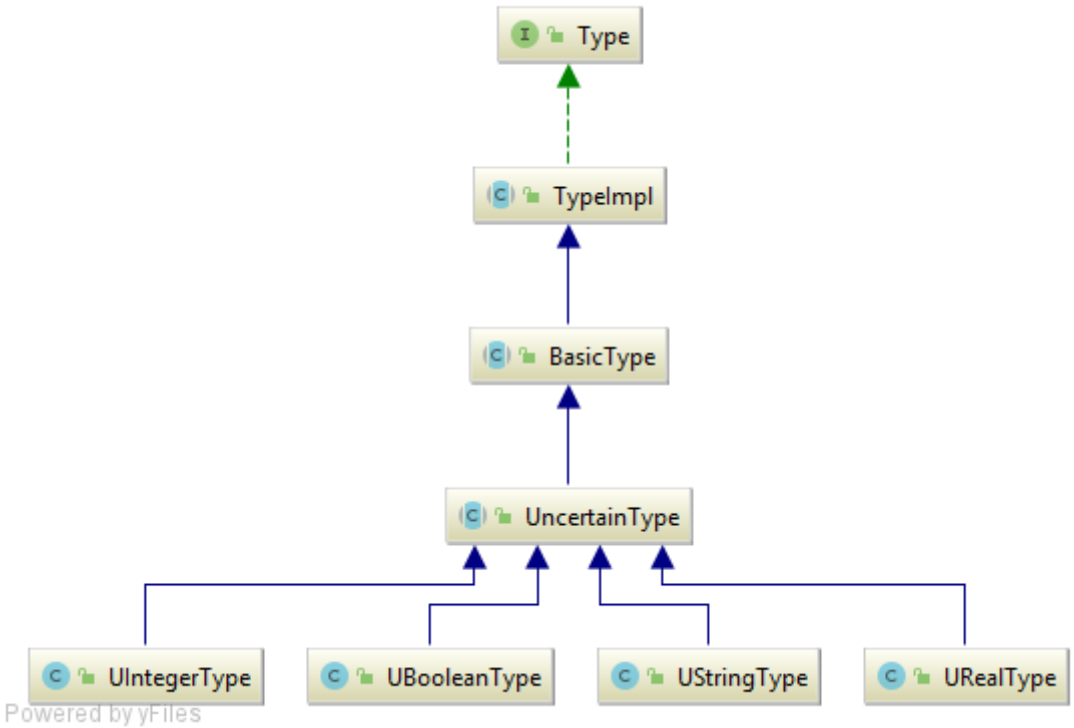

Figura 11: Jerarquía de tipos con incertidumbre

Como vimos en el apartado 4 – tipos. La interfaz *Type* define métodos para consultar el tipo de un objeto. Por lo tanto, al incorporar nuevos tipos, hay que incorporar también nuevos métodos para la consulta de éstos. Cuando hicimos estos nos dimos cuenta de que de

la interfaz *Type* dependen muchas partes de USE. Porque al modificarla, hubo que modificar otras clases.

Los métodos que se incorporaron fueron *isKindOfUInteger(), isTypeOfUInteger(), isKindOfUReal(), isTypeOfUReal(), isTypeOfUBoolean(), isKindOfUBoolean(), isKindOfUString(), isTypeOfUString()*.

4.3 Extensión de valores

En este apartado es dónde se encuentra una de las partes más importantes del proyecto o que encontramos más interesante. Porque es dónde se establecen qué valores tienen los tipos con incertidumbre, y como veremos a continuación, también sus operaciones. Empezaremos hablando de la aplicación del patrón *Adapter*, después hablaremos de las particularidades de la implementación, continuaremos con cómo se comparan los valores, ya que hay ciertas particularidades.

Aplicación del patrón *Adapter*

Aplicamos el patrón *Adapter* para resolver el problema planteado en el primer apartado 5.1. Para ello, se ha creado una clase adaptadora de cada tipo de datos de la librería. Por ejemplo, la clase correspondiente a UInteger en la librería, en use se llama UIntegerValue.

Al realizar operaciones, la librería devuelve valores de los tipos que ella utiliza. Por ejemplo, si ejecutamos *"UInteger.uEquals(q)"* devolverá un *UBoolean,* por lo tanto, hay que transformar este valor a *UBooleanValue*. De esta manera, desacoplamos el uso de la librería más allá de las clases adaptadoras. Esto permite qué si hay algún cambio en la librería, no se propague y esté centralizado en estas clases. Es más, en futuro se podría cambiar el comportamiento de la librería sin modificar la misma. Simplemente, derivando el patrón de diseño *Adapter* a *Proxie*, que son muy parecidos.

Implementación

Al igual que con los tipos, también hay que modificar la clase abstracta *Value* para añadir los métodos para identificar los nuevos tipos. En este caso no hubo que modificar tantas clases, puesto que los valores no tienen tantas dependencias.

Como hemos añadido nuevos tipos al sistema, necesitamos crear los métodos dentro de *Value* para consultarlo. Por lo que se han añadido estos métodos a la clase *Value*. Al igual que en los tipos, hemos creado una clase abstracta para agrupar todas las clases que

representan valores con incertidumbre, como se puede observar en la Figura 12. Esta solución aporta varios beneficios, uno de ellos es que podemos saber si una instancia corresponde a un valor con incertidumbre, que resulta útil al operar con estos valores, y la más obvia agrupar el comportamiento común. En este caso, los valores con incertidumbre tienen una forma más de compararse que los valores sin incertidumbre, por lo que se han añadido los métodos *uEquals()* y *uDistinct()* en este lugar. De los cuales el programador que tenga que implementar un nuevo valor con incertidumbre, tendrá que especificar el primero, ya que el segundo se ha puesto en función de este.

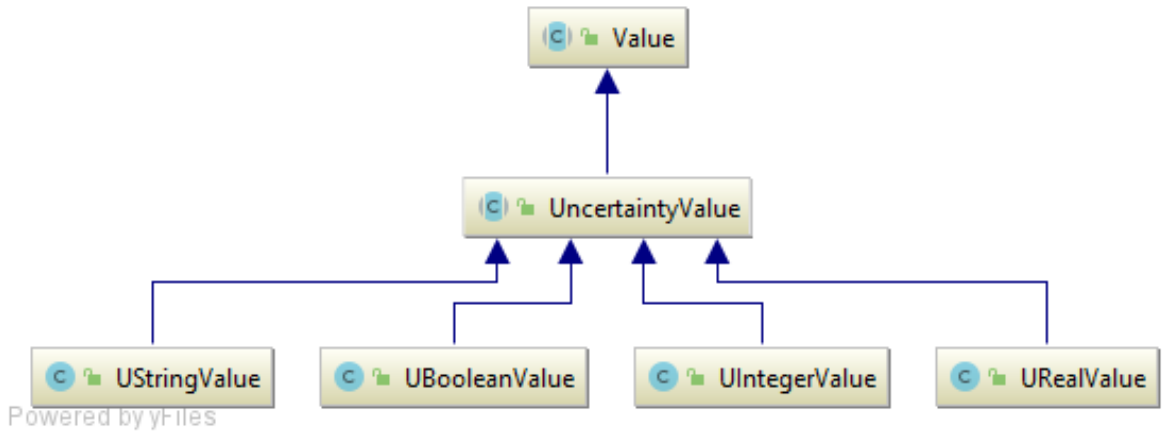

Figura 12: Jerarquía de valores con incertidumbre

Comparación entre valores

A agregar valores a la herramienta también tenemos que tener en cuenta la interoperabilidad. Es decir, que todos los valores sean operables entre los que ya había. Este requisito se corresponde más bien a la extensión de las expresiones, que es el lugar donde se operaran con ellos, pero a nivel de valores de definen cómo se comparan entre ellos. De modo, que teníamos que hacer que se comparen todos con todos.

Para implementar las comparaciones, había que implementar para cada valor con incertidumbre los métodos *compareTo(), equals()* y *hashCode()*.

En el primer caso no es muy útil para este proyecto, puesto que el resultado de tal operación sería booleano, y cualquier operación con un valor con incertidumbre debe dar como resultado un booleano con una cierta confianza (*UBoolean*). Como en la librería ofrecía esta funcionalidad, nos limitamos a enlazar este comportamiento con el de la librería. De modo, que cualquier valor que se le pase como argumento a esta función, se intenta elevar a un tipo con incertidumbre equivalente, y la función resulta en lo estipulado en la librería. Por ejemplo, en la expresión "1 < UReal(1, 2)", el constante "1" sería elevado a "UReal(1, 0)" y se llamaría a la función.

El caso de *equals()* es simple, dos instancias de *Value* son iguales si tienen el mismo estado. Pero hay que tener en cuenta que hay que elevar los valores al supertipo común. Aun así, encontramos ciertas dificultades al hacer estás comparaciones por la naturaleza del tipo básico de Java *double.* Por ejemplo, si en Java intentas ejecutar la expresión "1 – 0.8" dará como resultado "0.1999996..." y no "0.2". Este error fue captado por las pruebas unitarias y una vez conocido el problema, se solventó de la siguiente forma. Nos reunimos para decidir una cifra de precisión en las igualdades, que fue al décimo decimal. Para compararlos se halla el error relativo entre varias medidas, si no excede del umbral, entonces serán iguales, en caso contrario serán distintos.

Por último, tuvimos que implementar la *hashCode()* que es diferente al de la librería ofrecida. Puesto que internamente en USE los tipos de datos equivalente tenían que tener el mismo *hashcode* para el manejo de las colecciones. Por ejemplo, "2" tiene que tener el mismo *hash* que "2.0", al igual que "UReal(2, 0)". Por defecto, la implementación previa era usando la función de *hash* de los tipos *Double* de Java. Los valores con incertidumbre se pueden representar como una tupla (x, u) donde *x* es el valor *y* u la incertidumbre. Para el cálculo su cálculo se ha procedido como según la Figura 13 donde *hash()* corresponde a la función *hash* de Java para los tipos *double* para *UReal* y *UInteger*, en el caso de *UString* y *UBoolean* hash(x) se sería el hash de Java para *Boolean* y *String*.

$$hash(x, u) \begin{cases} hash(x) & \text{si } u = 0 \\ hash(x) * 7 + hash(u) & \text{en otro caso} \end{cases}$$

Figura 13: Función para el hash de valores con incertidumbre

De este modo, todos los valores escalares (que no tienen incertidumbre) de la aplicación tendrán el mismo *hash*. Siendo esta función equivalente a la función *equals()*.

4.4 Extensión de expresiones constantes

Para extender las clases constantes se ha tenido que crear una clase por cada tipo de dato con incertidumbre que se ha creado. Veremos qué clases se han creado, que estructura se de clases se ha creado y cómo se representan los valores a través de USE.

Implementación

La jerarquía de clases que se ha implementado se puede observar en la Figura 14. En ella podemos observar las cuatro clases de las que hablamos antes. Cada una de ellas debe de generar un valor. Para ello, simplemente hubo que crear el constructor correspondiente para cada caso e implementar el método *eval()* de cada expresión. En este método, aparte de generar el valor, se controla que éste sea correcto. Como ejemplo del que hablamos antes, no se puede crear un valor "UBoolean(true, 2.3)". Si esto ocurriese, se ha procedido a lanzar una excepción del tipo *ExpInvalidException.*

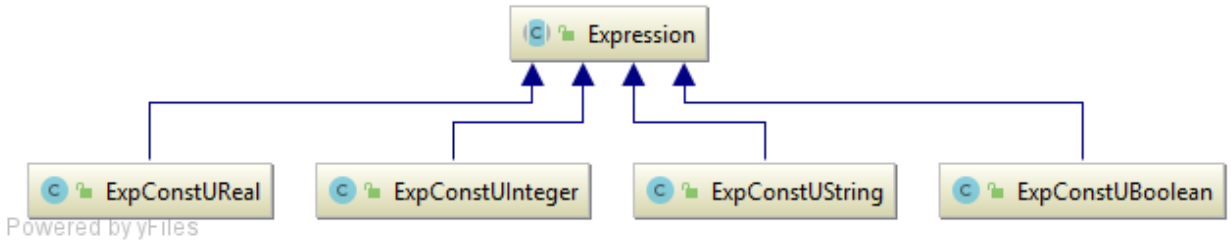

Figura 14: Jerarquía de expresiones constantes que se han implementado

Representación de los valores

Los creadores de USE especificaron cómo se debe implementar la representación de los valores. De hecho, no se hace mediante el método de Java por defecto *toString()*, si no que más bien, decidieron crear el suyo propio. El método se llama igual, pero recibe un objeto *StringBuilder* como argumento. Probablemente para construir de una forma más eficiente las expresiones complejas a la hora de presentarlas al usuario.

A parte de lo anterior, como las expresiones se construyen unas a partir de otras formando una estructura de tipo árbol, los creadores de USE usaron el patrón de comportamiento *Visitor.* Con él se consigue hacer la representación de las expresiones de forma limpia, sin "ensuciar" las nuevas clases de expresiones.

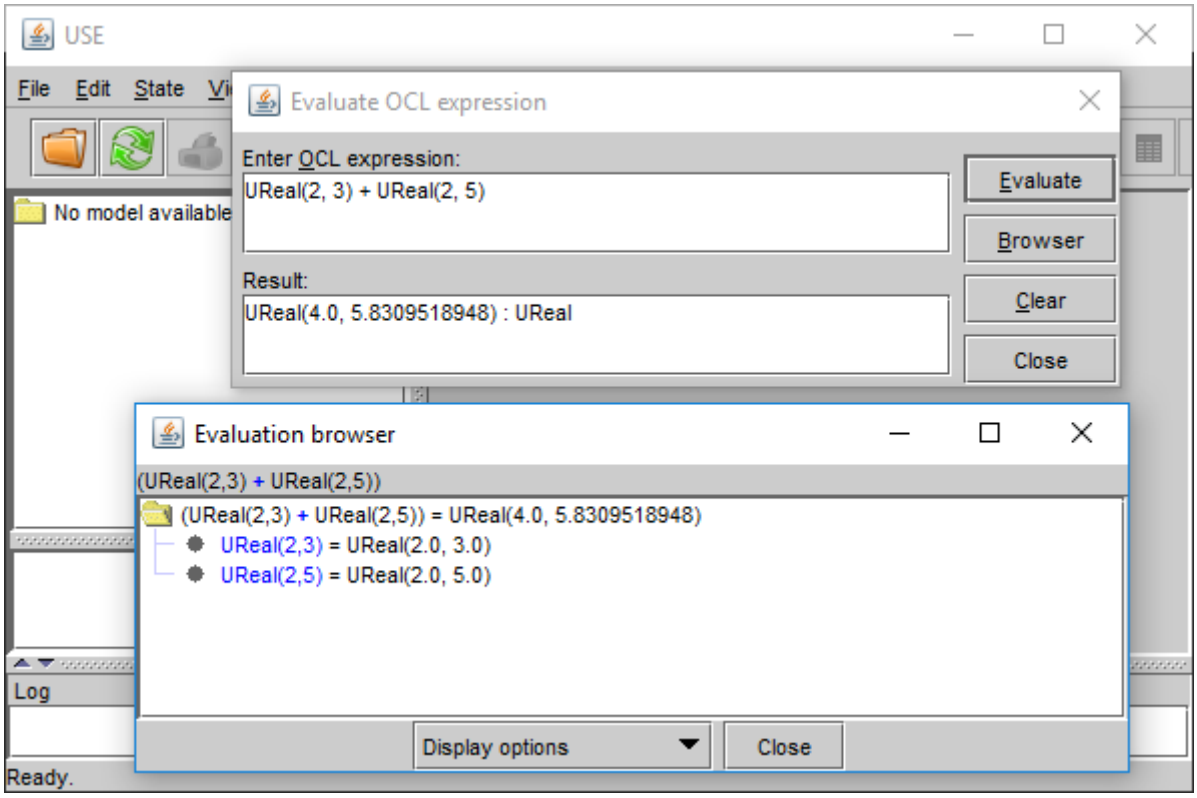

Figura 15: Ejemplo de representación de valores por medio de Visitor

Como ejemplo del resultado de esta forma de representar los valores se encuentra en la Figura 15. En ella se usa este comportamiento para construir el árbol que aparece en la ventana "*Evaluation Browser*" de la expresión introducida.

Para implementar esta funcionalidad que existía en la herramienta, a parte de la creación de las clases que anteriormente hemos hablado, hubo que modificar esta funcionalidad para aceptar las clases nuevas. Para ello, se tuvo que extender la interfaz responsable cual se encuentra en el mismo paquete de las expresiones y se llama *ExpressionVisitor*. Al ser una interfaz, existen varias clases a lo largo de la herramienta que la implementan, cómo la que se muestra en el ejemplo, u otras que usan este comportamiento para hacer medidas de análisis de cobertura, u transformar el árbol a otro formato como HTML. Por lo tanto, también hubo que retocar el comportamiento de estas clases.

4.5 Extensión de expresiones de operaciones estándar

La extensión de las expresiones de USE es sencilla, puesto que estaba hecho de una forma correcta y permite una fácil extensión del as mismas. Sin embargo, encontramos varios problemas del proyecto en sí, y no de implementación. De modo, que tuvimos que decidir qué hacer sabiendo que por un lado ganaríamos, pero perderíamos por el otro. En este apartado,

hablaremos de estas decisiones que tuvimos que acatar y de la implementación que se ha hecho.

Implementación

Para crear las expresiones de operaciones sobre los tipos nuevos de datos no hubo que hacer nada con respecto a la gramática. Puesto que también estaba preparada para estos cosos. Por lo tanto, lo único que se tuvo que hacer es crear una clase que agrupara estas operaciones de los tipos nuevos.

Siguiendo el ejemplo de las clases ya creadas en la aplicación, existe una clase que agrupa a las demás por tipo de dato. Es decir, existe una clase llamada *StandardOperationsBoolean* que contiene tantas clases internas como operaciones tiene los tipos booleanos en la aplicación.

De modo, que hicimos lo mismo, se crearon tres clases para agrupar a los tipos de datos nuevos; *StandardOperationsUBoolean, StandardOperationsUString, StandardOperationsUInteger* y *StandardOperationsUReal.* Estas clases no sólo se encargan de agrupar las operaciones, también las registran en la clase *OpGeneric.* Como los requisitos de este proyecto eran principalmente hacer operaciones en USE sobre estos tipos de datos, en ellas podemos encontrar prácticamente una clase por requisito funcional del proyecto.

Caso particular de las operaciones numéricas

En OCL existe un supertipo para los valores que son números y en nuestro caso, tenemos dos valores que se van a introducir que son subtipos de este tipo, *UReal* y *UInteger*. Para los números existía una clase que agrupaba a todas estas operaciones. Ya que todos los números se deben operar entre sí. Esta clase se llama *StandardOpertaionsNumbers.*

Se decidió que todas operaciones que tuviera que ser interoperable entre los distintos números se colocara en esta clase. Por lo tanto, ha sido modificada con respecto a su versión inicial. Puesto que antes esta clase sólo tenía que lidiar con *Integer* y *Real*, y ahora tiene que tratar con los tipos numéricos con incertidumbre. Por lo que han sido modificadas todas las operaciones de igualdad, comparación, y operaciones tal como suma, resta, multiplicación, etc.

Hasta aquí el cómo implementamos las expresiones dentro de USE, pero algunas de estas derivaron un poco desde la versión inicial de los requisitos. Esto sucedió al ver las operaciones en la práctica y tuvimos que tomar alguna que otra decisión acerca de su funcionamiento.

Deberíamos de hacer sobrecarga de operadores

Una cuestión que sucedió al principio del proyecto es qué operaciones de las impuestas por los requisitos se podían sustituir por un símbolo, y si debíamos de hacerlo. Esta pregunta surgió porque hay ciertas operaciones entre tipos donde surge cierta ambigüedad. Por ejemplo, hay dos formas de comparar los valores con incertidumbre, teniendo en cuenta la incertidumbre (uEquals()) y no teniéndola (equals) ¿a cuál de estas dos operaciones le asignamos el operador "="? La primera opción compara los dos valore devuelve un booleano con incertidumbre y el segundo un booleano, al ser dos tipos distintos, tenemos que tomar una decisión y ser consecuente con todas las operaciones que les ocurra lo mismo.

Tras varias reuniones sobre todas estas posibles asignaciones de símbolos, llegamos a una conclusión. Toda operación entre dos o más valores debe de dar como resultado el equivalente al supertipo común. Es decir, si se suman un valor con incertidumbre y uno que no, este último se elevará al supertipo común y éste será el tipo del resultado. Si, por el contrario, la operación fuera de comparación, el resultado sería *UBoolean*. En cierto modo, esto ya lo hacía la herramienta, porque la suma de un real y un entero es un real, y éste es su supertipo común.

Al aplicar esta decisión de diseño, ya no hay duda de a qué tipo de igualdad le asignaremos al símbolo "=". Puesto que el supertipo común entre un tipo sin incertidumbre y otro con, será de tipo *UBoolean* y este corresponde a la operación de igualdad *uEquals()*.

Esta decisión también se extiende a todos las operaciones de comparaciones. Para los símbolos <, >, <=, >=, <> se le han asignado las operaciones *lt(), gt(), le(), ge()* y *uDistinct()* respectivamente. Para las operaciones de suma, resta, multiplicación y división de los números también se le han asignado los símbolos de dichas operaciones. Al igual que para el tipo *UBoolean* se le ha asignado los símbolos *and, not, implies, or* y *xor*. Que son interoperables con el tipo booleano clásico de OCL.

Como todo, esta solución no es perfecta y tienes varias implicaciones en la compatibilidad hacia atrás de lo que ya existía en USE. Por ejemplo, las condiciones de if-else no reconocen el tipo *UBoolean*, al igual que cualquier otro aspecto de USE que reconociera *Boolean*. Si el usuario intentará establecer en la condición de un if-else una expresión que derivará en *UBoolean*, tendría un error de compilación. Para solventar este problema, se pensó en que siempre se podía "bajar" el tipo de *UBoolean* a *Boolean* por medio del a función *UBoolean.toBoolean()* y así permitir el tratar con estos tipos en las condiciones de los if-else, u en otra parte.

Al aceptar que el tipo de igualdad "=" por defecto sería *uEquals()*, hay que tener en cuenta que, por el funcionamiento de esta, dos valores serán iguales sí la probabilidad de serlo es mayor o igual a 0.5. Por lo que hay que lidiar con esta propagación implícita que se hará a través del modelo definido por el usuario. Creemos que es un hecho del cual el usuario debe de ser consciente por si se produjeran situaciones extrañas. Al igual que en el problema anterior, si se quiere tener un mayor control del umbral de igualdad entre dos valores, se puede encapsular la condición de la que se esté tratando y ejecutar la operación *UBoolean.toBooleanC().*

Caso de varias igualdades al inicio del proyecto

Las formas de igualar dos tipos con incertidumbre en un principio eran diversas y han ido cambiado la idea sobre ellas. Al inicio del proyecto se recogieron como requisitos las funciones *uEquals(), identical()* y el símbolo "=" correspondía a una operación que correspondía si dos valores con incertidumbre eran indistinguibles. Esto sucedía si uno era equivalente a otro. Con el cambio de la sobrecarga de operadores comentado anteriormente, se perdió el interés en este operador, porque se argumentó que no añadía información que no se pudiera obtener con los otros dos métodos de igualación.

El método *identical()* sufrió un cambio de nombre a *equals()*. El motivo fue porque tiene el mismo comportamiento que el método *equals()* de Java. Es decir, compara si ambos tienen el mismo estado. Llevándonos esto al terreno de este proyecto, dos valores con incertidumbre A = (x, u) y B = (x', u') son iguales si, y solo si, x = x' y u = u'. Por lo tanto, dándole este cambio de nombre y suponiendo que el usuario objetivo es un programador, reconocería al instante qué hace este método.

Con respecto a *uEquals* nada cambió, esta operación compara si dos valores con incertidumbre son iguales con una confianza del 50%. Simplemente como se ha comentado en el apartado anterior, se le asignó el símbolo "=".

4.6 Extensión de expresiones de colecciones

A diferencia de las operaciones estándar, las colecciones están implementadas de un modo muy diferente al de las expresiones estándares. No están agrupadas como ellas, ni tienen su propio paquete para distinguirlas de las demás. Si no que más bien, todas estas expresiones heredan de la clase *ExpQuery* que contiene todos los métodos para cada una de las operaciones que existen en USE sobre OCL. Aquí se encuentra la implementación de *forAll, exists, one*, etc.

Una de las cosas interesantes y de las que se ha aprendido mucho es que, aunque a veces en el código de USE haya distintas formas de implementar las funcionalidades que quisieron introducir, se aprende de cada una de ellas y da que pensar. Por ello, creemos que el código de USE es bastante educativo y muchas veces te hace pensar.

Como dependencias de la clase *ExpQuery* están los cuatro tipos de colecciones que hay en OCL. Estas cuatro colecciones son *Bag, Sequence, Set* y *OrderSet* que se pueden encontrar en el paquete *org.tzi.use.util.collections*.

Evolución de las operaciones de colecciones

Las operaciones de colecciones fueron lo último en implementarse en el proyecto. Por lo que al llegar al aparte de la implementación se hicieron nos hicimos unas preguntas distintas a las de partida ¿Qué hacemos con aquellas operaciones de colecciones que devuelven un booleano?

Por ejemplo, estamos de acuerdo en el que la expresión "{1, 2, 3, 4}->forAll(e | e>=0)" el resultado será "true". Sin embargo, qué pasará si en cambio introducimos una expresión como la siguiente "{UReal(1, 0.5), UReal(2, 0.25)

}->forAll(e | e >= 0)". Ya que esta operación propagará por cada valor la expresión "e >= 0" y está dará como resultado *UBoolean* como sabemos, por la sobrecarga de operadores. De modo que ¿deberíamos bajar el tipo a *Boolean*?¿ o permitir que las funciones devuelvan *UBoolean* y *Boolean* según la naturaleza de los argumentos?

El debate aquí está en la compatibilidad hacia atrás, estamos modificando las funciones básicas de OCL en USE, y con el tiempo que hubo, no se sabía si iba a tener consecuencias con respecto a la compatibilidad hacia atrás de la herramienta. Aunque nos parecía una muy buena idea. Por lo tanto, concluimos en hacerlo de manera selectiva. Decidimos que donde más atractivo sería esta funcionalidad sería para las operaciones *forAll, exists, includes, includesAll, excludes, excludesAll* y *isUnique*. Todas estas funciones en un principio devolvían *Boolean* y solamente su versión con incertidumbre devolvía *UBoolean*. Para cada una, la versión con incertidumbre tenía el prefijo de *"u"* para distinguirlas.

Sin embargo, para las operaciones que devuelven como resultado otra cosa distinta de un valor booleano, decidimos dejarlas tal y como estaban en un principio. Ya que el cambio anterior podría entrar en conflicto con modelos de incertidumbre ya planteados anteriormente. De modo, que todas las operaciones *select, reject, any* y *count* tienen su versión de incertidumbre *uSelect, uSelectC, uReject, uRejectC, uAny, uAnyC, uCount* y *uCountC*.

Cortocircuito para forAll y exists

Al explorar el código de *forAll* y *exists* nos dimos cuenta de que no implementaban el cortocircuito en las condiciones cuando éstas se evaluaban. Hecho que no pudimos resistirnos a preguntarnos el por qué. Motivo de esto tuvimos una reunión y se aclaró que no siempre es recomendable utilizar el cortocircuito en la resolución de las expresiones booleanas.

Explorando las opciones, al parecer la implementación del cortocircuito puede derivan en "efectos laterales". Es decir, comportamientos del código no controlado que pueden derivar en un error futuro. Por ejemplo, si tenemos dos variables reales x e y, la siguiente condición con cortocircuito no fallaría " y <> 0 and x / y". Puesto que si se ejecuta primero "y <> 0" y es evaluado a false, no se ejecutará "x / y". Pero y si tenemos lo siguiente "condición A and hazAlgo()", donde *condicionA* es una expresión booleana y *hazAlgo()* es una función. Un efecto lateral no controlado podría suceder si la *condicionA* fuera falsa y la función tuviera que ejecutarse por cualquier motivo. Por ejemplo, porque tuviera que reservar algún recurso, etc.[SCE]

Esta información nos pareció interesante e implementamos estos métodos sin cortocircuito. A diferencia de las operaciones *UBoolean.and()* and *UBoolean.or()*, estas se hicieron antes que estas operaciones y si tenían cortocircuito previamente. Por lo que se respetó esta decisión de los creadores.

Implementación

Con todo lo anterior explicado, la implementación que se hizo está recogida en la Figura 16. Se puede observar que todas heredan de *ExpQuery* y como ya se dijo antes, todas las funciones para ejecutar la lógica de la operación se encuentran en ella. Por lo que lo que se tuvo que hacer es crear o modificar estos métodos y crear una clase hija que llamara a estos.

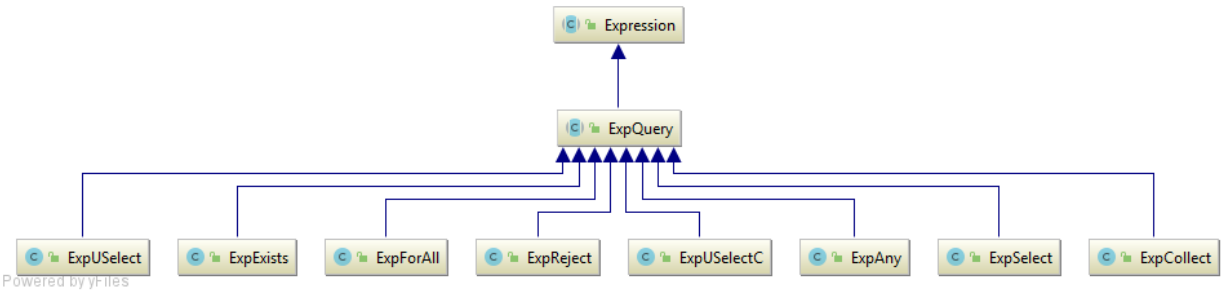

Figura 16: Jerarquía de operaciones con colecciones

4.7 Extensión de la gramática

Hasta aquí hemos hablado de casi todos los puntos para extender los tipos de datos en USE a falta de este. Tenemos que modificar la gramática para que el compilador reconozca estos tipos y sepa cómo tiene que construirlos. Como veremos a continuación, se ha hecho una modificación en la gramática y se han añadido nodos al árbol AST.

Extensión

Para extender la gramática de OCL sólo tenemos que modificar el archivo base de esta gramática, *org.tzi.use.parser.base.OCLBase.gpart*. Como vimos en el estudio inicial, después se generarán los ficheros necesarios en tiempo de compilación a raíz de este.

La gramática de OCL dentro de USE tiene como base una expresión. De esta, se van definiendo de manera recursiva las demás, respetando la precedencia por niveles. Para añadir los nuevos literales con incertidumbre, tenemos que modificar una de las hojas del árbol gramatical. En concreto, esta hoja se llama literal y se puede observar su estructura en la Figura 17.

Para extender este nodo introducimos recursividad a las nuevas declaraciones. Todo valor con incertidumbre tiene la estructura "UValue(x, u)" donde x e u pueden ser expresiones. Si fuera el caso de UReal, deben de estar admitidos en el lenguaje los siguientes casos : "UReal(1, 0)", "UReal(1.0, 0.2)", "UReal(1 + 2, 2 / 1 * 29)", "UReal(2, UReal(2, 32).value())", etc.

La opción por la que se optó se puede observar en Figura 18. Se puede observar que existen 4 nuevas reglas una para cada tipo de valor nuevo, y que aceptan una expresión de suma, excepto en el caso de *UBoolean*. Aceptan una expresión de suma porque la operación con menos preferencia que fuera numérica. Ocurre lo mismo con *UBoolean,* pero con las expresiones booleanas y el operador *implies*.

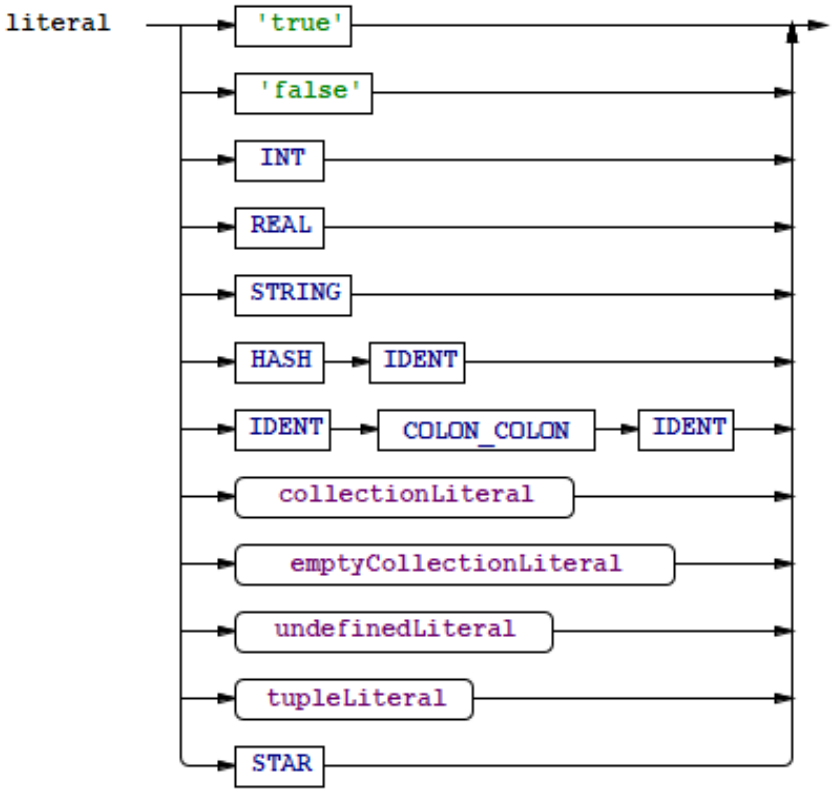

Figura 17: definición de literales antes de los valores con incertidumbre

De acuerdo, esta solución funciona, pero ¿entonces la gramática acepta la cadena "UBoolean(2 + 4, 2.2)"? La respuesta es que sí, y no. El compilador acepta tal cadena, pero el nodo que general la misma, correspondiente al árbol AST no lo acepta y lanza una *SemanticException*. De modo que de esta forma es como se ha lidiado con estos casos.

¿Por qué se optó por esta solución y no se modificó la gramática para que el compilador detectara este error? La respuesta es que esta solución era más sencilla que la formulada en la pregunta. Antes de atacar al problema, se estudió el código de USE y se observó que la aplicación ya lo hace en según qué casos. Con esta solución la gramática también se queda más limpia. No se han añadido más reglas de las que había, solamente se ha añadido recursividad.

Por lo tanto, en la implementación se añadieron los nodos al árbol AST *ASTUBooleanLiteral, ASTURealLiteral, ASTUIntegerLiteral* y *ASTUStringLiteral* que se encuentran en el paquete *org.tzi.use.parser.ocl.*

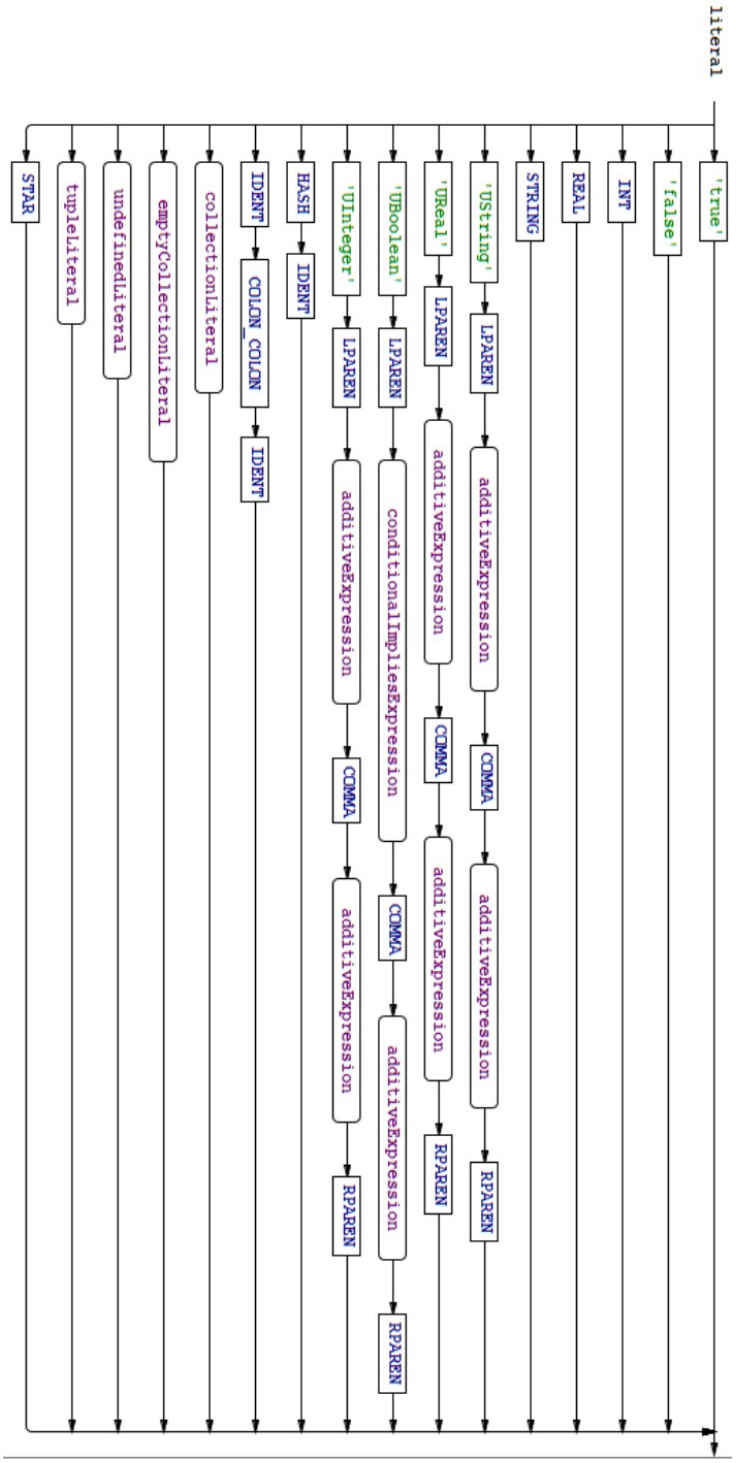

Figura 18: Definición de literales con valores con incertidumbre

5

Validación y pruebas

En este capítulo entraremos en detalles en qué estrategia se ha seguido para realizar las pruebas y por qué. Veremos la metodología que se ha seguido para realizar cada una de ellas, esta parte la ilustraré con un ejemplo real del proyecto.

También explicaremos cómo se han implementado las pruebas y unos detalles que ayudaron a que el proyecto tomara un ritmo mucho más productivo, con relación a la generación de pruebas.

5.1 Técnicas y procedimientos que se han usado

Para realizar las pruebas de cada requisito se ha seguido una metodología vista en la universidad que consiste en identificar de prueba independientes (*Independent Test Functionality*, ITF de aquí en adelante*)*. Por ejemplo, si tenemos que probar la definición de un tipo *UBoolean(x, c)*, donde *x* es "true" o "false"y *c* es un número entre [0-1], un ITF podría ser "comprobar que la probabilidad *c* está entre 0 y 1".

Normalmente se fijan varios ITF para cada uno de los requisitos, esto también es muy útil, porque de este modo agrupan los casos de prueba. Una vez identificados los ITF, por cada uno se identifican los posibles casos que pudieran ocurrir. Siguiendo nuestro ejemplo anterior, puede que la confianza (c) sea menor que 0, sea 0, se encuentre entre 0 y 1, sea 1 o sea mayor que 1. Una vez identificados los casos que se pueden dar, se particionan y se crean los casos de prueba. En este ejemplo solamente hemos tenido en cuenta la confianza, pero

debemos recordar que el usuario puede meter cualquier tipo de datos, ya que la entrada es en tipo texto.

Cuando se tienen todas las posibilidades o situaciones en las que hemos pensado que se podrían dar, se hace recuento de cuántos casos de prueba serían necesarios hacer. El número de casos de prueba crece exponencialmente, puesto que se tienen que combinar todos los casos en los que hayamos pensado. Muchas veces, este número ascendía a más de 100 pruebas para una funcionalidad. Lo que se volvía inviable en tiempo, se tuvo que tomar otras técnicas.

Una vez se tienen la definición de los ITF y nos encontramos en una situación de un número de pruebas impracticable. Se tiene que optar por técnicas de reducción de número de pruebas. Para ello, se diseñaba un diagrama de actividad en UML en el que se representaba el posible flujo de acciones que el programa debería de hacer para conseguir que la funcionalidad sea correcta. Con este diagrama, se sacaban los casos de prueba siguiendo los caminos designados por los nodos de decisión. La guía básica de cuántos casos de prueba tenemos que extraer está designado por el número de complejidad axiomática del diagrama.

En ambos casos, puede que no se halle el 100% de cobertura de código probado. Si nos encontramos en este caso, es necesario crear más pruebas. Para ello, tenemos que identificar más reglas y hacer más combinaciones de situaciones que se puedan dar (que probablemente, debido a que las pruebas crecerán de forma exponencial, se tendrá que hacer un diagrama y extraerlas de ahí). Una vez tenemos las nuevas reglas, se retoca el diagrama de flujo y se extraen las pruebas. Esto se hace hasta que se consigue el 100% de la cobertura. Podemos observar el proceso descrito en la Figura 19.

Este tipo de técnica se encuentra dentro de las pruebas de “caja negra” del software. Ya que no se tiene en cuenta nada del código, solamente tiene como objetivo que una funcionalidad se cumpla acorde a su especificación. No se han realizado pruebas de “caja blanca”. Puesto que no se disponía del tiempo suficiente y éstas se suelen hacer en cuando el proyecto se encuentra en un estado “Alpha”. En el que el cliente dispone del software, y mientras el equipo de programación se encuentra realizado este tipo de pruebas y resolviendo incidencias. Este punto del proyecto no se alcanzó, debido a que fue justo de tiempo, y la implementación se terminó cerca de la fecha de entrega.

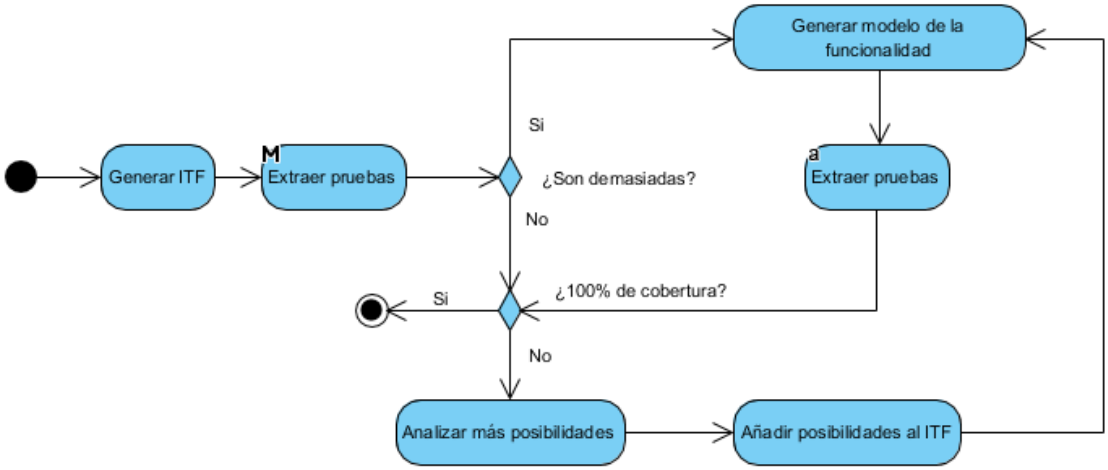

Figura 19: Proceso de generación de pruebas de caja negra

De todos modos, creemos que se ha optado por una buena decisión a la hora de practicar las pruebas de caja negra. Ya que éstas están pensadas para descubrir fallos en los que no se había pensado en un principio, debido a su naturaleza combinatoria. Además, uno de los objetivos del proyecto era probar la librería ofrecida por los tutores del proyecto. La cual, ha sido probada a través de la extensión de esta aplicación.

A lo largo del proyecto la construcción del proyecto se ha tenido que mantener la librería por parte de los tutores que corregían las incidencias, que han ido surgiendo a raíz de las pruebas hechas por el autor de este proyecto.

5.2 Implementación de las pruebas

Una vez descrito el proceso de generación de pruebas, tenemos que hablar de dónde se encuentra éstas y por qué se ha organizado de ese modo.

Como ya hemo dicho antes, las pruebas que se han realizado son unitarias y de sistema. Siempre se ha tenido presente que las pruebas sean lo suficientemente descriptivas y los errores se puedan depurar para encontrar antes el problema. De modo, que lo que más se ha primado ha sido la mantenibilidad de estas.

Las pruebas dentro de USE se han hecho en dos fases. La primera como pruebas unitarias, y la segunda como pruebas de sistema. En este proyecto, los casos de prueba de la primera están contenidos dentro de la segunda. Es decir, muchas de las pruebas de sistema ya han sido probadas por las pruebas unitarias.

El motivo de por qué se realizó de esta manera es porque mediante las pruebas unitarias se intentaba detectar los errores más probables. Con el objetivo de aprovecharse del

framework de JUnit y realizar la depuración. Dado que, si éstas fueran de sistema, se obtendría un error por la terminal indicando el error y su depuración sería más tediosa.

Las pruebas de sistema se encuentran los ficheros escritos con la definición de la gramática de pruebas, y se creó una clase para que leyera estos ficheros, interpretara las pruebas y las ejecutara. El problema es que estos ficheros tienden a ser muy largos, de hecho, suelen tener entre 1000-2000 líneas. Por lo tanto, si fallara un caso de prueba a mitad del fichero, sería muy difícil poner un punto de ruptura y explorar el caso de prueba si fuera necesario.

Este mecanismo de pruebas de sistema ya estaba impuesto por los creadores de USE, y tiene sus puntos fuertes y débiles. Ya hemos hablado de un punto débil, que es el problema para corregir errores. Pero en cambio, estos ficheros están escritos en un lenguaje más cercano al humano. Por lo que el programador puede escribir expresiones complejas y ejecutarlas en el compilador. Por ejemplo, el código de prueba equivalente en Java a la expresión " (UReal(2, 3) * UReal(3, 4)) * UReal(4, 5) = UReal(2, 3) * (UReal(3, 4) * UReal(4, 5))" tendría del orden de más de 50 líneas de código.

Por lo tanto, tomamos la decisión de generar las pruebas unitarias mediante las técnicas vistas anteriormente, generar con ellas las pruebas unitarias, copiar estas mismas en un fichero escritas en el lenguaje de la gramática de pruebas, y por último añadir más pruebas. Estás últimas pruebas solían ser sobre todo de propiedades que una operación debía cumplir. Por ejemplo, podrían ser que la suma de dos números reales fuera asociativa, o que fuera conmutativa. También se reservaban para esta fase pruebas que no se podían ejecutar como pruebas unitarias. Por ejemplo, aquellas que lo que probaban era un fallo de compilación. Por ejemplo, "UBoolean('booleano', 1).

Hasta aquí, hemos hablado de forma general de cómo se han escrito las pruebas en dos fases. Pero hay casos en los que solamente se han ejecutado pruebas en la fase 1. Esto es debido a que USE ya tenía pruebas en un principio y una de las cosas que hemos tenido presente cada vez que extendíamos esta herramienta, es que queríamos "mimetizarnos" con lo que ya había hecho, no cambiarlo. En la extensión de tipos y valores, existían pruebas unitarias por parte de los creadores.

El caso de los tipos es especial, porque se tiene que hacer una estructura de herencia entre los tipos. Por ejemplo, Real es un subtipo de UReal, por lo que, al modificar este comportamiento muchas de las pruebas de USE fallarán. De modo que se tuvo que identificar los casos de prueba de los tipos y modificarlos. A parte de añadir nuevos con los requisitos impuesto en el proyecto.

En el caso de los valores los creadores tenían pruebas de tipado y de comparaciones entre ellos. Estos no han fallado durante la extensión de la herramienta, pero si se han extendido. Se identificó los casos de prueba que perseguían las personas que crearon la herramienta y se adaptaron a los nuevos valores. Esto era útil porque uno de los requisitos no funcionales que teníamos en mente era la interoperabilidad. Además, cuanto más probáramos como casos de prueba unitaria, más fácil sería corregir los errores. Por el motivo que hemos hablado antes acerca de que éstas son más fáciles de corregir que las de sistema. En otros casos, se ha seguido con la descripción ofrecida anteriormente.

De acuerdo ¿dónde se encuentran estas pruebas? Debido a que todas las pruebas que se han realizado son para validar la funcionalidad de la extensión de OCL, podemos encontrarlas en el paquete de pruebas *org.tzi.use.uml.ocl*. Aquí encontrarás las clases encargadas de probar las expresiones, tipos y valores de los tipos de datos añadidos.

Las pruebas de sistema están recogidas en el paquete *org.tzi.use.parser.uncertainty.* En el podremos encontrar tanto la clase que se encarga de leer estas pruebas, como las pruebas de sistema que se han hecho para los 4 tipos de datos que se han extendido.

5.3 Optimización de tiempo

Un problema que teníamos con la implementación de los casos de prueba era el tiempo que se necesitaba para realizarlo. Cada vez que se terminaba un requisito, se anotaba cuánto tiempo se había tardado en hacerlo. Por ejemplo, se tardaba más de 3 horas en hacer un requisito junto a sus pruebas. Esto junto con el tiempo limitado que se disponía para la realización del proyecto suponía un riesgo para el mismo.

El principal problema era la escritura de estos casos de prueba. Puesto que se tenían que escribir en dos partes, como pruebas unitarias (en Java) y como pruebas de sistema (con la gramática de pruebas). Las pruebas se escribían en hojas de Excel, por lo que se escribió un programa en Java que pasara texto desde las hojas Excel a un lenguaje y otro.

El funcionamiento con el programa era el siguiente, se copiaba y pegaba en el programa las columnas de Excel (Figura 20), y se escribía cómo se tenían que procesar estos datos. Para hacer referencia a cada columna se utilizaba el símbolo $ seguido del número de la columna (ej: $1 sería la columna 1, $2 la segunda ,etc). Una vez establecidos estos dos requisitos, la herramienta nos generaba el texto que teníamos que copiar y pegar como código de Java u el que fuera. Un ejemplo de uso lo podemos ver en la Figura 21, a la izquierda está la herramienta y la a derecha de ella, el código pegado de la caja de texto inferior.

	A		B		Resultado	
Numero	**x**	**u**	**x'**	**u'**	**x''**	**u''**
UC3218-01	-9	0	-9	0	0.0	0.0
UC3218-02	-5	0	-5	3	0.0	3.0
UC3218-03	-4	0	2	0	-6.0	0.0
UC3218-04	-10	0	4	1	-14.0	1.0
UC3218-05	-9	9	-9	0	0.0	9.0
UC3218-06	-2	3	-2	4	0.0	5.0
UC3218-07	-6	2	5	0	-11.0	2.0
UC3218-08	-2	3	4	4	-6.0	5.0
UC3218-09	0	0	0	0	0.0	ls
UC3218-10	0	0	0	4	0.0	4.0
UC3218-11	0	0	6	0	-6.0	0.0
UC3218-12	0	0	7	3	-7.0	3.0
UC3218-13	0	4	0	0	0.0	4.0
UC3218-14	0	4	0	3	0.0	5.0
UC3218-15	0	4	1	0	-1.0	4.0
UC3218-16	0	4	2	3	-2.0	5.0
UC3218-17	9	0	9	0	0.0	0.0
UC3218-18	5	0	5	3	0.0	3.0
UC3218-19	4	0	8	0	-4.0	0.0
UC3218-20	10	0	10	12	0.0	12.0
UC3218-21	9	5	9	0	0.0	5.0
UC3218-22	2	3	2	4	0.0	5.0
UC3218-23	6	1	4	0	2.0	1.0
UC3218-24	2	3	5	4	-3.0	5.0

Figura 20: Casos de prueba de resta de UReal

Esta herramienta es muy rutinaria y no está pulida ni mucho menos. Pero ayudó a reducir la implementación de un requisito de más de 4 horas a 1-2 horas. Por lo que fue un soplo de aire fresco en el desarrollo del proyecto. Incrementando muchísimo el rendimiento en implementación de requisitos.

Figura 21: Ejemplo de generación de pruebas para *UReal.minus()* con la herramienta de generación.

6

Conclusiones

En este capítulo exponen las conclusiones fruto del desarrollo de la extensión, así como experiencias del autor sobre el trabajo realizado, así como qué se ha aprendido del mismo. Al final, también se exponen posibles extensiones de este proyecto.

6.1 Estudio de USE

El estudio inicial fue la parte más difícil de realizar de este proyecto. En esta fase se disponía de mucha incertidumbre, en el sentido en el que USE es muy grande y muchos días parece que el proyecto no avanza. Aunque esta es solo una ilusión del comienzo de un proyecto, ya que gracias a este estudio se acortó mucho el tiempo de implementación de la extensión. Además, se ha aprendido muchísimo de cómo está hecho USE. Muchas veces he encontrado interesante cómo se está hecha una funcionalidad y he indagado más por curiosidad. Esto es debido a que siempre he tenido curiosidad sobre cómo aplicar patrones de diseño o investigar cómo modelar un sistema. Al estar el modelo hecho de los tipos de datos, valores y expresiones, y su interacción entre ellos, pude ver una solución al modelado de este problema.

Como resumen, el estudio de use fue frustrante al inicio porque no se sabía donde se encontraban las partes del código que se tenían que modificar, pero cuando ya se supo, fue muy gratificante.

6.2 Compiladores

Tampoco sabía inicialmente nada de compiladores, solamente aquello que hemos dado en la universidad, y esto es relativo a la teoría sobre las gramáticas, pero no cómo llevar a cabo su implementación, ya que esto no forma parte de la Ingeniería del Software. Para ello tuve a mano el libro de referencia de ANTLR [ANTLR] y como ejemplo el código de USE para aprender sobre compiladores. Como conclusión sobre esta experiencia he decir que ahora pienso que se deberían de usar más las gramáticas para toda interacción textual entre el humano y la máquina. Ya que el lenguaje resultante de una gramática suele ser más cercano al humano, y si disponemos de herramientas como ANTLR, deberíamos utilizarlas para conseguir este objetivo.

El hecho de que este proyecto tuviera el requisito de tener que usar compiladores fue uno de los motivos para escogerlo. Porque es uno de los temas que no hemos dado en la carrera y había investigado por mi cuenta. De modo que opté por elegir este proyecto y profundizar este tema.

6.3 Desarrollo y pruebas

El desarrollo de la aplicación no fue una gran dificultad más allá de las decisiones que se tuvieron que tomar ante las dificultades encontradas. Por lo general, los requisitos que tomaban más tiempo eran aquellos de creación de nuevos tipos, ya que por la propia estructura de la creación implementar nuevas expresiones era una tarea sencilla.

El principal problema ha estado en el desarrollo de las pruebas unitarias y de sistema, que incrementaban mucho el tiempo implementación de un requisito. Muchas veces, este tiempo de desarrollo de pruebas era del orden de 4 veces el tiempo de implementación del código. Esto en las fases iniciales del proyecto, donde se tenía muy poco conocimiento de cómo estaba hecha la herramienta, llegaba a ocupar de 4 horas a 8 horas para un requisito.

Sin embargo, a parte de las dificultades expuestas, he encontrado muy útil el trabajar con las pruebas unitarias y de sistema junto a la metodología TDD. Más de una vez, he modificado código que introducía errores en USE y estos han sido detectadas por las pruebas de regresión. Si no hubiera seguido esta metodología estos errores podrían haber pasado desapercibidos.

Este proyecto me ha ayudado a mejorar las estrategias para probar el software. Conforme se ha ido a avanzando en el proyecto, se ha ido tendiendo a una implementación de pruebas que fuera mantenible. Porque, por ejemplo, cuando se corregía un error que se

hubiera detectado en la librería que calcula las operaciones de incertidumbre, normalmente tenía efectos laterales y fallaban después otros casos de prueba. A veces, era por una mala especificación de las pruebas y se tenían que modificar los casos de prueba. De modo que, si se consiguen que éstas sean mantenibles, este proceso será mucho más rápido.

Referencias

[CPS] Broy, M.: Challenges in modeling Cyber-Physical Systems. In: Proc. of ISPN'13, pp. 5–6. IEEE (2013)

[OCL] Büttner, F., Gogolla, M.: On OCL-based imperative languages. Sci. Comput. Program. 92, 162–178 (2014)

[SOIL] Büttner, F, Gogolla, M.: *Modular Embedding of the Object Constraint Language into a Programming Language*. Universidad de Bremen [en linea][fecha de consulta : 24/06/2019]. Disponible en : http://www.db.informatik.uni-bremen.de/publications/Buettner_2011_SBMF.pdf

[TDD] C. JORGENSEN, Paul. Test-Driven Development. En: Software Testing: A Craftman's Approach. 4Th edition. Roca Raton: CRC Press, 2014, pp. 375-385. ISBN 978-1-4665-6069-7

[USE] Gogolla, M., Büttner, F., Richters, M.: USE: A UML-based specification environment for validating UML and OCL. Sci. Comp. Prog. 69, 27–34 (2007)

[IoT] Greengard, S.: The Internet of Things. MIT Press (2015)

[ANTLR] Parr, Terence: *The Definitive ANTLR Reference Building Domain-Specific Languages*. The Pragmatic Bookshelf. Raleigh, North Carolina Dallas, Texas. 2007. ISBN-10: 0-9787392-5-6

[SCE] Short-circuit evaluation – Wikipedia [en línea][fecha de consulta: 24/06/2019]. Disponible en https://en.wikipedia.org/wiki/Short-circuit_evaluation

Apéndice A

Manual de Instalación

Requerimientos

Para ejecutar el proyecto necesitarás tener instalado el JDK de Java a partir de la versión 1.7.0 (http://java.oracle.com) Puesto que USE está implementando en este lenguaje de programación. A parte de esto, no se necesitará mucho más, porque todo lo que necesita para funcionar viene en la carpeta suministrada en el CD.

Al ser una herramienta de código libre, el programa viene junto su código fuente. De modo, que usted es libre de compilar el código o modificarlo. Para compilar el código simplemente sitúese en el directorio de USE desde su consola de comandos y ejecute “ant”. Nótese que, en este caso será necesario instalar el programa Apache Ant 1.6.0 o una versión mayor (https://ant.apache.org/), para interpretar la compilación de USE.

A parte de lo anterior, si usted también quiere ejecutar todas las pruebas que vienen incluidas en USE. Necesitará estar en un entorno UNIX o tener muchas de sus utilidades por consola de comandos. Puesto parte de las pruebas están escritos en otros lenguajes como Perl, de modo que se necesitará utilidades como make, perl, sed, etc.

Como requisito mínimo es tener la máquina virtual de java (JDK), si usted la tiene, puede ejecutar el programa con el fichero *bin*/*start_user.bat* o *bin/use* si se encuentra en un entorno UNIX.

Apéndice B
Requisitos

En este apéndice se muestran los requisitos funcionales que se recogieron en la fase de análisis. Cuando se habla de un tipo numérico, se refiere al conjunto de tipos {UReal, UInteger, Integer, Real}.

R1 Se podrá crear, operar y transformar tipos UInteger

R1.1 Se podrá crear un tipo UInteger

R1.2 Se podrá operar con tipos UInteger

R1.2.1 Se podrá obtener un valor absoluto de un tipo UInteger (abs())

R1.2.2 Se podrá obtener el valor contrario a un tipo UInteger (neg())

R1.2.3 Se podrá obtener la potencia de un tipo UInteger (power())

R1.2.4 Se podrá obtener la raíz cuadrada de un tipo UInteger (sqrt())

R1.2.5 Se podrá sumar un UInteger y otro tipo numérico (+)

R1.2.6 Se podrá restar un UInteger y otro tipo numérico (-)

R1.2.7 Se podrá multiplicar un UInteger y otro tipo numérico (*)

R1.2.8 Se podrá dividir un UInteger y otro tipo numérico (/)

R1.2.9 Se podrá obtener la división entre un tipo entero y un UInteger (div())

R1.2.10 Se podrá obtener el módulo entre un tipo entero y un UInteger (mod())

R1.2.11 Se podrá obtener el valor de un UInteger (value())

R1.2.12 Se podrá obtener la incertidumbre de un UInteger (uncertainty())

R1.2.13 Se podrá modificar el valor de un UReal (setValue())

R1.2.14 Se podrá modificar la incertidumbre de un UInteger (setUncertainty())

R1.3 Se podrán comparar un tipo UInteger y otro numérico (=,<>,equals,<,>,<=,>=)

R1.3.1 Se podrá comprobar si un tipo UInteger es igual a otro tipo numérico. (equals())

R1.3.2 Se podrá comprobar si un tipo UInteger es igual a otro tipo numérico con una cierta confianza (=)

R1.3.3 Se podrá comprobar si un tipo UInteger es distinto de otro tipo numérico con cierta confianza (<>).

R1.3.4 Se podrá comprobar si un tipo UInteger es menor que otro tipo numérico (<)

R1.3.5 Se podrá comprobar si un tipo UInteger es menor o igual que otro tipo numérico. (<=)

R1.3.6 Se podrá comprobar si un tipo UInteger es mayor que otro tipo numérico. (>)

R1.3.7 Se podrá comprobar si un tipo UInteger es mayor o igual que otro tipo numérico. (>=)

R1.4 Se podrá transformar un tipo UInteger a otro tipo numérico.

R1.4.1 Se podrá transformar un tipo UInteger a Integer.

R1.4.2 Se podrá transformar un tipo UInteger a UReal.

R1.4.3 Se podrá transformar un tipo UInteger a Real.

R1.4.4 Se podrá transformar un tipo UInteger a String.

R2 Se podrá crear, operar y transformar tipos UBoolean

R2.1 Se podrá definir un tipo UBoolean

R2.2 Se podrá operar con tipos UBoolean(not, and, or, xor, implies, equivalent)

R2.2.1 Se podrá aplicar la operación “not” a un tipo UBoolean

R2.2.2 Se podrá aplicar la operación “and” a un tipo UBoolean

R2.2.3 Se podrá aplicar la operación “or” a un tipo UBoolean

R2.2.4 Se podrá aplicar la operación “xor” a un tipo UBoolean

R2.2.5 Se podrá aplicar la operación “implies” a un tipo UBoolean

R2.2.6 Se podrá aplicar la operación “equivalent” a un tipo UBoolean

R2.2.7 Se podrá obtener al valor del tipo UBoolean (value())

R2.2.8 se podrá modificar el valor del tipo UBoolean(setValue())

R2.2.9 Se podrá obtener el valor de confianza de un tipo UBoolean (confidence())

R2.2.10 Se podrá modificar la confianza de un tipo UBoolean (setConfidence())

R2.3 Se podrán comparar tipos UBoolean (=,<>, equals, equalsC)

R2.3.1 Se podrán comprobar si dos tipos UBoolean son iguales. (equals())

R2.3.2 Se podrá comprobar si dos tipos UBoolean son iguales con una cierta confianza. (equalsC())

R2.3.3 Se podrá comprobar si dos tipos UBoolean tienen incertidumbres equivalentes (=)

R2.3.4 Se podrá comprobar si dos tipos UBoolean tienen incertidumbres distintas (<>)

R2.4 Se podrán transformar tipos UBoolean

R2.4.1 Se podrá transformar un tipo UBoolean a String (toString)

R2.4.2 Se podrá transformar un tipo UBoolean a Boolean (toBoolean)

R2.4.3 Se podrá transformar un tipo UBoolean a Boolean acorde a una confianza (toBooleanC)

R3 Se podrá crear, operar y transformar tipos UReal

R3.1 Se podrá definir tipos UReal

R3.2 Se podrá operar con tipos de datos UReal

R3.2.1 Se podrá obtener un valor absoluto de un tipo UReal (abs())

R3.2.2 Se podrá obtener el valor contrario a un tipo UReal (neg())

R3.2.3 Se podrá obtener el valor redondeado a la baja de un tipo UReal (floor())

R3.2.4 Se podrá obtener el valor redondeado de un tipo UReal (round())

R3.2.5 Se podrá obtener el valor inverso de un tipo UReal (inv())

R3.2.6 Se podrá obtener la potencia de un tipo UReal (power())

R3.2.7 Se podrá obtener la raíz cuadrada de un tipo UReal (sqrt())

R3.2.8 Se podrá obtener el seno de un tipo UReal (sin())

R3.2.9 Se podrá obtener el coseno de un tipo UReal (cos())

R3.2.10 Se podrá obtener la tangente de un tipo UReal (tan())

R3.2.11 Se podrá obtener el arcoseno de un tipo UReal (asin())

R3.2.12 Se podrá obtener el arco-coseno de un tipo UReal(acos())

R3.2.13 Se podrá obtener el arco-tangente de un tipo UReal (atan())

R3.2.14 Se podrá sumar un UInteger y otro tipo numérico (+)

R3.2.15 Se podrá restar un UInteger y otro tipo numérico (-)

R3.2.16 Se podrá multiplicar un UInteger y otro tipo numérico (*)

R3.2.17 Se podrá dividir un UReal y otro tipo numérico (/)

R3.2.18 Se podrá obtener el mínimo de dos UReal (min())

R3.2.19 Se podrá obtener el máximo de dos UReal (max())

R3.2.20 Se podrá obtener el valor de un tipo UReal (value())

R3.2.21 Se podrá modificar el valor de un tipo UReal (setValue())

R3.3 Se podrán comparar un tipo UReal y otro numérico (=,<>,equals,<,>,<=,>=)

R3.3.1 Se podrá comprobar si un tipo UReal es igual a otro tipo numérico (equals)

R3.3.2 Se podrá comprobar si un tipo UReal es igual a otro tipo numérico con una cierta confianza (=)

R3.3.3 Se podrá comprobar si un tipo UReal es distinto de otro tipo numérico con cierta confianza (<>).

R3.3.4 Se podrá comprobar si un tipo UReal es menor que otro tipo numérico. (<)

R3.3.5 Se podrá comprobar si un tipo UReal es menor o igual que otro tipo numérico. (<=)

R3.3.6 Se podrá comprobar si un tipo UReal es mayor que otro tipo numérico. (>)

R3.3.7 Se podrá comprobar si un tipo UReal es mayor o igual que otro tipo numérico. (>=)

R3.4 Se podrá transformar tipos de datos UReal

R3.4.1 Se podrá transformar un tipo UReal a Integer.

R3.4.2 Se podrá transformar un tipo UReal a UInteger.

R3.4.3 Se podrá transformar un tipo UReal a Real.

R3.4.4 Se podrá transformar un tipo UReal a String.

R4 Se podrá crear, operar y transformar tipos UString

R4.1 Se podrá crear un tipo UString

R4.2 Se podrá conocer el tamaño con incertidumbre de un dato UString (size)

R4.3 Se podrá conocer la letra en una determinada posición de UString (at)

R4.4 Se podrá conocer la posición de comienzo de una subcadena dentro de otra cadena con incertidumbre (indexOf)

R4.5 Se podrán operar con tipos UString

R4.5.1 Se podrá concatenar dos UString (+)

R4.5.2 Se podrá obtener una subcadena de un tipo UString(substring())

R4.5.3 Se podrá obtener a la cadena del tipo UString (value())

R4.5.4 Se podrá modificar la cadena del tipo UString (setValue())

R4.5.5 Se podrá obtener la confianza sobre la cadena de un tipo UString (confidence())

R4.5.6 Se podrá modificar la confianza sobre la cadena de un tipo UString (setConfidence())

R4.6 Se podrán comparar una UString con otra cadena (=,<>,equals, uEqualsIgnoreCase, <,<=,>,>=)

R4.6.1 Se podrá comprobar si una UString es igual a otra cadena. (equals)

R4.6.2 Se podrá comprobar si una UString es igual a otra cadena con una cierta confianza (=)

R4.6.3 Se podrá comprobar si una UString es igual a otra cadena con incertidumbre, ignorando si está en mayúscula o en minúscula (uEqualsIgnoreCase)

R4.6.4 Se podrá comprobar si una UString es distinta a otra cadena (<>)

R4.6.5 Se podrá comprobar si una UString es menor que otra cadena (<)

R4.6.6 Se podrá comprobar si una UString es menor o igual a otra cadena (<=)

R4.6.7 Se podrá comprobar si una UString es mayor que otra cadena (>)

R4.6.8 Se podrá comprobar si una UString es mayor o igual a otra cadena (>=)

R4.7 Se podrá transformar un UString a otro tipo.

R4.7.1 Se podrá transformar UString a String.

R4.7.2 Se podrá transformar UString a otra UString en mayúsculas (toUpperCase)

R4.7.3 Se podrá transformar UString a otra UString en minúscula (toLowerCase)

R4.7.4 Se podrá transformar UString a Integer (toInteger)

R4.7.5 Se podrá transformar UString a Boolean (toBoolean)

R4.7.6 Se podrá transformar UString a UBoolean (toUBoolean)

R4.7.7 Se podrá transformar UString a Real (toReal)

R5 Se dará soporte a las operaciones de colección a los tipos de datos nuevos.

R5.1 Se podrá ejecutar con incertidumbre la función *forAll* de OCL

R5.2 Se podrá ejecutar con incertidumbre la función *exists* de OCL

R5.3 Se podrá ejecutar con incertidumbre la función *collect* de OCL

R5.4 Se podrá ejecutar con incertidumbre la función *includes* de OCL.

R5.5 Se podrá ejecutar con incertidumbre la función *excludes* de OCL.

R5.6 Se podrá ejecutar con incertidumbre la función *includesAll* de OCL.

R5.7 Se podrá ejecutar con incertidumbre la función *excludesAll* de OCL.

R5.8 Se podrá ejecutar con incertidumbre la función *isUnique* de OCL.

R5.9 Se podrá ejecutar con incertidumbre la función *sum* de OCL.

R5.10 Se podrá ejecutar una versión de la función *select* de OCL con incertidumbre (uSelect)

R5.11 Se podrá ejecutar una versión de la función *select* de OCL con incertidumbre y una cierta confianza (uSelectC)

R5.12 Se podrá ejecutar una versión de la función *reject* de OCL con incertidumbre (uReject)

R5.13 Se podrá ejecutar una versión de la función *reject* de OCL con incertidumbre y una cierta confianza (uRejectC)

R5.14 Se podrá ejecutar una versión de la función *count* de OCL con incertidumbre (uCount)

R5.15 Se podrá ejecutar una versión de la función *count* de OCL con incertidumbre y una cierta confianza (uCountC)